

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

FREERUMMI: Juego de fichas numeradas

Daniel Chaves Vázquez

11 de mayo de 2011



ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

FREERUMMI: Juego de fichas numeradas

- Departamento: Lenguajes y Sistemas Informáticos
- Directores del proyecto: Juan Manuel Dodero Beardo y Manuel Palomo Duarte
- Autor del proyecto: Daniel Chaves Vázquez

Cádiz, 11 de mayo de 2011

Fdo: Daniel Chaves Vázquez

Agradecimientos

Me gustaría dedicar este texto a todas las personas que han hecho posible que hoy pueda escribirlo, en especial a mis padres, a mi hermana y a Lourdes, así como me gustaría agradecer a Manuel Palomo, director del proyecto, por la ayuda y los consejos prestados.

Licencia

Este documento ha sido liberado bajo Licencia GFDL 1.3 (GNU Free Documentation License). Se incluyen los términos de la licencia en inglés al final del mismo.

Notación y formato

Al escribir este texto se ha empleado un conjunto de convenios tipográficos para facilitar la lectura y comprensión del documento:

- Cuando nos refiramos a un programa, utilizaremos la notación: *emacs*, *Gimp*.
- Cuando nos refiramos a un comando de consola:

```
Comando en consola
```

- Cuando nos refiramos a las distintas secciones que componen el juego: **Jugar**, **Ver**.
- Cuando nos refiramos a una función de un lenguaje o un fichero de código: `ficha.h`, `escogerficha()`.

Índice general



Indice de figuras

Indice de tablas

Capítulo 1

Introducción

1.1. Objetivos

El ocio está cada vez más presente en la vida cotidiana y la informática no es ajena a dicho cambio: cada vez aparecen nuevos dispositivos que nos permiten tener experiencias impensables hasta hace poco con un ordenador o disfrutar de nuestros juegos preferidos en cualquier parte.

El objetivo de este Proyecto Fin de Carrera es implementar el juego de mesa Rummikub®. Rummikub®, es un juego que combina lógica y estrategia creado en la década de 1930 por Ephraim Hertzano. En él pueden participar de 2 a 4 jugadores, y combina elementos del Rummy, Dominó, Mah-jongg y Ajedrez.

Se intenta mantener una máxima fidelidad al juego original en cuanto a movimientos, reglas y elementos se refiere. Se desea crear un videojuego con una interfaz fácilmente comprensible con una amplia gama de opciones que resulten interesantes para los jugadores. Además estará liberado bajo licencia GPL, ya que a día de hoy existe alguna versión para ordenador pero ninguna libre.

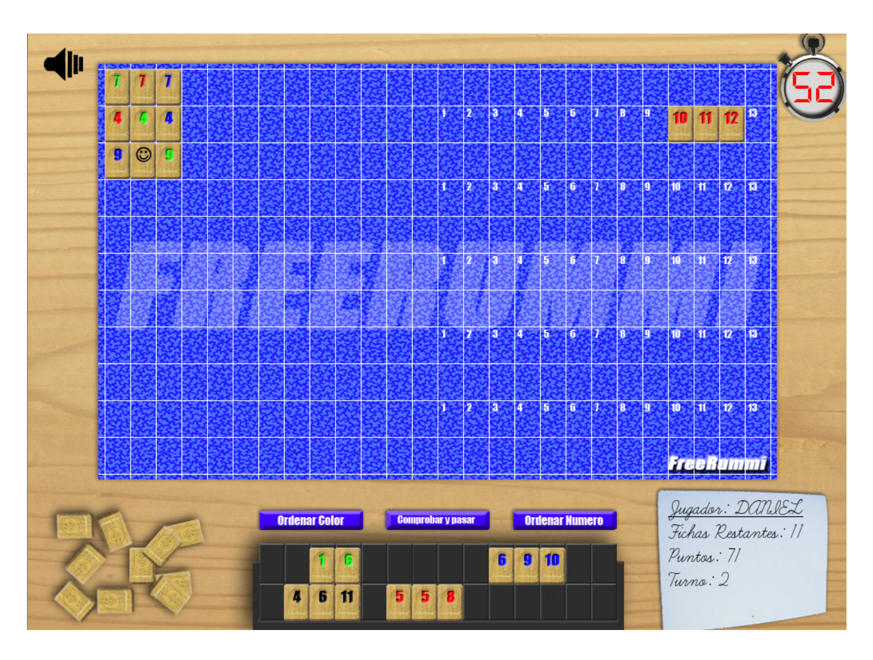


Figura 1.1: Captura de una partida de FreeRummi

1.2. Estructura del documento

El primer capítulo de este documento es en el que nos encontramos actualmente. Contempla los objetivos de mi Proyecto Final de Carrera así como la estructura de este documento.

En el segundo capítulo se verán los conceptos básicos a tener en cuenta: qué es Rummikub®y que herramientas he utilizado para su desarrollo y adaptación a mi proyecto.

En el tercer capítulo se expone la planificación que he seguido durante el periodo de desarrollo del proyecto, incluyendo un diagrama de Gantt y los diferentes incrementos por los que ha pasado el proyecto.

Los tres siguientes capítulos abarcan el proceso de ingeniería del software: especificación de requisitos, análisis y diseño.

En el séptimo capítulo se habla sobre la implementación del videojuego, las técnicas y herramientas usadas, así como las diferentes estrategias que se han seguido para realizarlo.

El octavo capítulo repasa las diferentes pruebas que se han realizado al código.

En el noveno capítulo presento las conclusiones e impresiones que he tenido a lo largo del desarrollo del proyecto.

En el último capítulo se incluye distintos apéndices tales como el manual de instalación, el manual de usuario y a bibliografía.

Al ser un producto de Software Libre se incluye la licencia en la que se basa la documentación del proyecto.

Capítulo 2

Conceptos básicos

En este segundo capítulo se presenta la historia y reglas del juego *Rummikub*®, así como las herramientas usadas para el desarrollo de mi versión particular del mismo.

2.1. Rummikub

FreeRummi es la creación de una versión libre (liberado con licencia GPL) del juego de mesa Rummikub®, en el que se intenta mantener una máxima similitud al juego original, con una amplia gama de opciones.

Según wikipedia [15], Rummikub®, fue creado en los años 1930 por un inventor rumano, después emigrado a Israel, llamado Ephraim Hertzano. Después Hertzano lo exportó a Europa oeste y América.

El juego se compone de 106 fichas, 104 numeradas y 2 comodines. Las fichas están numeradas del uno al trece y en cuatro diferentes colores (negro, rojo, azul y amarillo). Cada jugador tiene un soporte para guardar sus fichas de forma que el oponente no las vea, el soporte es un atril parecido al que se usa en el Scrabble.

En primer lugar, las fichas son puestas en una bolsa o saco pequeño, cada jugador saca 14 fichas al azar para jugar. Para decidir quién empieza primero, todos sacan una ficha y luego la devuelven. Empieza quien sacó el número más alto, en el caso de un comodín se pueden aplicar reglas propias, o según las reglas oficiales su valor es de 30, otra opción es sacar otra ficha.

Para empezar a poner los jugadores deben tener uno o mas grupos que sumen 30 puntos o más , los puntos son tomados según el número, en el caso de usarse un comodín su valor es 30. En el caso de no tener una combinación de 30 puntos o más el jugador debe robar una ficha pasar el turno. La imagen que sigue muestra un soporte con más de 30 puntos repartidos en tres grupos de fichas 2.1:



Figura 2.1: Ejemplo de soporte con más de 30 puntos

Una vez puesta su primera combinación,el jugador puede poner cualquier otra sin importar su puntuación o aportar fichas a las ya existentes, en el caso de no poner ninguna ficha en su turno,deberá robar una ficha del montón y pasar el turno.

Existen 2 tipos de grupos: los conjuntos, y la escalera. Ambas combinaciones deben contener como mínimo tres fichas,no debes poner, ni debe existir sobre la mesa un grupo con menos de tres fichas.

- Los Conjuntos están compuestas por 3 o 4 fichas del mismo número, pero de distinto color, por ejemplo 2.2:

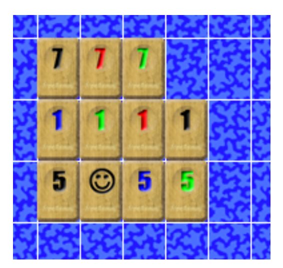


Figura 2.2: Ejemplos de posibles conjuntos

- Las Escaleras están compuestas por 3 o más fichas del mismo color y con un valor numérico continuo, por ejemplo 2.3:



Figura 2.3: Ejemplos de posibles escaleras

El juego continua hasta que a uno de los jugadores se le agotan las fichas entonces debe gritar Rummikub para declararse ganador. Cuando un jugador gana los demás jugadores deben sumar la puntuación de las fichas que quedaron en su soporte (puntuación del juego). El comodín tiene un valor de 30 puntos. La puntuación de los perdedores es negativa y la del ganador es la suma de todos los demás jugadores, pero positiva.

En el *manual de usuario de FreeRummi* (apéndice B), se incluye más información a cerca de las reglas de juego, ejemplos de manipulación, estrategias a seguir, etc.
Existe una versión oficial que permite jugar online a Rummikub®, [11] pero es una versión limitada a unas pocas partidas al día y que al ser privativa no tiene opciones de expansión.

2.2. Tecnologías implicadas

En esta sección se presentan las herramientas que he utilizado para el desarrollo del proyecto.

Para el desarrollo del juego he utilizado:

- *C++*: Lenguaje de programación orientado a objetos, multiplataforma. Todo el videojuego se ha realizado con este lenguaje.
- *libSDL*: Librería gráfica libre basada en C/C++, multiplataforma. Es la base gráfica y de sonido de mi videojuego. Es libre y está compuesta por los subsistemas de vídeo, sonido, eventos, red, etc.
- *emacs*: Editor de textos libre. Es el editor utilizado para la codificación de todo el juego, así como para la memoria.
- *Adobe®Photoshop®CS2*: Herramienta de diseño gráfico de la marca Adobe. La he utilizado para la realización de todos los elementos gráficos del juego.
- *GIMP*: Es una herramienta diseño gráfico completa, simple y libre. La he utilizado complementariamente a Photoshop®, como herramienta para crear los elementos gráficos del juego.

Por otro lado, he utilizado las siguientes herramientas libres, para distintos fines:

- *LaTeX*: Sistema de composición de textos. Herramienta usada para la creación de la memoria.
- *Planner*: Herramienta para el desarrollo de diagramas de Gantt y de tareas.
- *Doxygen*: Herramienta para generar la documentación del código del juego.
- *Dia*: Programa para generar diagramas. Utilizado en el análisis y diseño del proyecto.

Capítulo 3

Planificación

Dado que se ha partido de cero en la realización de este proyecto y que en principio no se conocen todas las funcionalidades que tendrá el sistema, lo más adecuado es utilizar el **modelo incremental**.

3.1. Incrementos

Se comenzará analizando los requisitos básicos del sistema, y a partir de ahí se empezará a desarrollar el programa por fases, evaluando en cada incremento que nuevas funcionalidades son necesarias para completar el juego. De esta forma, no se tratará como un todo, si no que serán distintas piezas que se van encajando.

En los siguientes apartados se explican los principales incrementos que se han llevado a cabo para la realización de *FreeRummi*.

3.1.1. Incremento 1: Selección de herramientas

En el primer incremento se ha estudiado el tiempo que se estimará necesario para la realización del proyecto, así como las herramientas que serán necesarias para el desarrollo (entorno de programación, lenguaje, imágenes, sonidos, ...).

He establecido un horario de trabajo de 10 a 14 y de 17 a 20 de lunes a viernes. La forma de trabajar ha sido realizando el análisis y el diseño de lo que iba necesitando y a continuación implementándolo, de forma que iba variando las tareas y hacía el trabajo más ameno. Igualmente no he realizado todas las imágenes y sonidos del juego de una vez, si no que los he ido haciendo e introduciendo a medida que los iba necesitando.

Decidí que el programa se codificaría en *C++*, ya que es el lenguaje que más he usado durante la carrera. Para ello me he ayudado de referencias como el libro *Fundamentos de C++* [5] o sitios web como *Cplusplus reference* [3].

Para el tema de los gráficos y el sonido me decanté por *libSDL*. Antes de esa decisión me tomé un tiempo para investigar algunas otras herramientas para programar con gráficos y con sonido, pero por toda la documentación [1] y tutoriales que tiene, además de por ser multiplataforma, me decanté por la anteriormente dicha *libSDL*.

Estudié la forma de documentar el proyecto y de hacer la memoria: para documentar el código me decanté por *Doxygen* [1], ya que conocía dicha herramienta de un curso que hicimos en la asignatura Diseño de Videojuegos, además de ser una de las mejores herramientas para generar documentación de manera automática. Para la memoria, no tenía claro como estructurarla, así que decidí ir haciendo un borrador a medida que iba avanzando. Cuando necesité estructurar toda la memoria, el director del proyecto me recomendó que usara una plantilla en LaTeX[9] que ha creado un compañero de la titulación y el libro *LaTeX: una imprenta en sus manos* [12] de tal forma que solo tuve que ir pasando y redactando el borrador, según la estructura que me daba la plantilla.

3.1.2. Incremento 2: Organización de las partes básicas del juego

En este incremento decido qué elementos son fundamentales para el juego, centrándome en la **partida** que es pilar central de mi proyecto.

Estos elementos son: **Fichas**, el saco o **Monton** de fichas, atriles o **Soportes** y un **Tablero**. Una vez decididos dichos elementos se estudió como se representarían en memoria, encontrándonos con las siguientes clases básicas para poder jugar:

- Clase `Ficha`: Una ficha se caracteriza por su número y su color.
- Clase `Monton`: El conjunto de todas las fichas es el Montón.
- Clase `Soporte`: Las fichas que cada jugador tenga se guardarán en su Soporte.
- Clase `Tablero`: Los conjuntos o escaleras de fichas que pueda formar pues las pondrá en el Tablero.

En las primeras versiones del juego, su ejecución se llevaba a cabo mediante consola 3.1, para comprobar el correcto funcionamiento del motor de manera independiente a su representación gráfica: crear las fichas, desordenar el montón y permitir extraer fichas de él, colocar las fichas en el soporte y permitir ordenarlas, insertar y extraer fichas, poder colocar conjuntos en el tablero, y lo más complicado: realizar todas las comprobaciones para que en el tablero no se pudieran insertar nada más que conjuntos y escaleras de fichas válidos.

[1] *Doxigen* es un generador de documentación para C++, C, Java, Objective-C, Python, IDL (versiones Corba y Microsoft) y en cierta medida para PHP, C# y D. Dado que es fácilmente adaptable, funciona en la mayoría de sistemas Unix así como en Windows y Mac OS X. La mayor parte del código de Doxygen está escrita por Dimitri van Heesch. Doxygen es un acrónimo de dox(document) gen(generator), generador de documentación para código fuente.

```
Fichas restantes: 14
1 - VERDE 1
2 - VERDE 10
3 - AZUL 1
4 - AZUL 4
5 - AZUL 8
6 - AZUL 11
7 - AZUL 12
8 - AZUL 13
9 - ROJO 2
10 - ROJO 6
11 - ROJO 8
12 - ROJO 10
13 - ROJO 11
14 - ROJO 12

MENU
1 Robar
2 Manipular fichas
3 Salir del juego

Eliga opcion:
```

Figura 3.1: Primera versión de FreeRummi en consola

Una vez realizado este incremento, tenia una base firme sobre la que ir añadiendo funcionalidades.

En cuanto a esfuerzo y tiempo, es una de las partes que más me costó, debido a que había que realizar muchas pruebas, para tener la seguridad de que a medida que iba avanzando, esta parte no me diera problemas.

3.1.3. Incremento 3: Introducción de gráficos y sonido

En este incremento comienzo a introducir gráficos para la partida, así como manejadores de evento, para poder jugar con el ratón.

Se incluyen las siguientes clases básicas para el sistema de vídeo y audio:

- Clase `Iniciado`: Esta clase inicia SDL.
- Clase `Video`: Esta clase es la encargada de controlar el subsistema de video de SDL.
- Clases `Musica` y `Efecto` : Estas clases se encargan de la banda sonora y de los efectos de sonido respectivamente, juntas controlan el subsistema de audio de SDL.

- Clase `Imagen`: Esta clase es la encargada de cargar y manipular una imagen del juego en la superficie de SDL.

- Clase `Fuente`: Esta clase es la encargada de cargar y manipular los textos que aparecen en la superficie de SDL.

Para un mayor orden incluí un archivo de cabecera `constantes.h`, el cual contiene las constantes para ubicar cada elemento en su sitio y así ser más fácil modificar su posición en el caso de que haya cambios
En este incremento me fue de gran ayuda el Tutorial de libSDL [2], realizado en 2008 por un compañero de la Universidad de Cádiz, así como el libro *Programación de videojuegos con SDL para Windows y Linux* [13].

3.1.4. Incremento 4: Diseño e implementación de la clase Partida

Una vez finalicé todas las clases bases para jugar y las clases para manejar SDL, empecé a añadir funcionalidades a la clase `Partida`. Este es el incremento que más tiempo y esfuerzo me llevó, junto con el primero ya que en este incremento se añadieron: eventos de teclado y ratón, cronometro, puntuaciones de los jugadores, etc.

Tenía claro desde un principio que el juego iba a ser completamente manejado por ratón. Debido a la dinámica del juego, si hubiese optado por el teclado, hubiese perdido mucha jugabilidad.
Para las distintas funcionalidades de la partida implementé las siguientes clases:

- Clase `Puntero`: Para el manejo de los eventos del ratón, la cual informa de en qué posición se produjo un cierto evento y con que botón se produjo.

- Clase `Cronometro`: Algo fundamental en un juego por turnos. Se trata de un temporizador de un minuto, el cual se puede pausar y reanudar.

- Clase `Jugador`: Esta clase es la encargada de gestionar los jugadores que se crean para jugar una partida. Es una de clase compleja en comparación con `Puntero` y `Cronometro`, ya que trabaja con ficheros y permite crear, cargar, guardar, borrar y actualizar los datos de un jugador.

Además de las nuevas creaciones, tuve que modificar y añadir algunas funciones a las clases base para poderlas relacionar con las clases de manejo de SDL. También se crearon y añadieron los elementos gráficos que faltaban así como el sonido del juego. A todo esto se une una gran cantidad de tiempo en probar cada funcionalidad que iba añadiendo.

Una vez finalizado este incremento ya tenía una partida en la que podían jugar de 2 a 4 jugadores.

3.1.5. Incremento 5: Diseño e implementación de menús

En este incremento añadí el menú principal y submenús con la clase `Menu`. En un primer momento el juego solo tenía un menú principal con las opciones:

- **Jugar**: Comenzaba una partida de cuatro jugadores.

- **Salir**: Finalizaba la ejecución del juego y volvía a Linux.

Una vez implementé y probé el funcionamiento de dichos menús, añadí a la opción **Jugar** un submenú en el que se proponen al jugador dos tipos de partida distintos:

- **Simple**: Se trata de una partida de entrenamiento para un solo jugador contra el ordenador.
- **Concurso**: Se trata de una serie de partidas entre jugadores.
- **Volver**: Vuelve al menú principal.

Finalmente añadí otra opción más al menú principal, la opción **Jugadores**, la cual permite la gestión de los jugadores. Esta opción desemboca en otro submenú con las opciones:

- **Ver**: En este apartado se observan las estadísticas de los jugadores.
- **Borrar**: En este apartado se pueden borrar jugadores del sistema.
- **Volver**: Vuelve al menú principal.

A modo de resumen el esquema de menús y submenús de *FreeRummi*, es el siguiente:

- **Jugar**
  - Simple
  - Concurso
  - Volver
- **Jugadores**
  - Ver
  - Borrar
  - Volver
- **Salir**

Este incremento no fue muy costoso, debido a que una vez que tuve el primer menú, los demás se hacían de forma similar por lo que no invertí mucho tiempo ni esfuerzo.

3.1.6. Incremento 6: Diseño e implementación del gestor de partida

En este incremento se añadieron los apartados **Simple** y **Concurso** del menú **Jugar** por medio de la clase `GestorPartida`.

Se incluyó la opción de elegir el número de jugadores a participar en la partida así como la creación/carga de los mismos.

3.1.7. Incremento 7: Diseño e implementación del gestor de jugadores

En este incremento se añadieron los apartados **Ver** y **Borrar** del menú **Jugadores** por medio de la clase `GestorJugadores`.

Se pueden crear tantos jugadores como el usuario desee y de cada jugador se guardarán los siguientes datos:

- Nombre
- Puntuación general

- Partidas ganadas
- Partidas perdidas
- Partidas abandonadas

Además por medio de los datos anteriores se obtendrán los siguientes derivados:

- Porcentaje de partidas ganadas
- Porcentaje de partidas perdidas
- Porcentaje de partidas abandonadas
- Rango del jugador: *Novel, Medio, Avanzado, Experto*. Este dato se obtendrá en función de la puntuación general del jugador: si tiene menos de 250 puntos, será *Novel*; si tiene entre 250 y 500 puntos, será *Medio*; si tiene entre 500 y 1000 puntos, será *Avanzado*; y si tiene más de mil puntos será *Experto*.

3.1.8. Incremento 8: Diseño e implementación de los sistemas Expertos

En este último incremento se añade la clase `IA`, la cual se encarga de gestionar y manipular los sistemas expertos que se creen para la partida simple. Para ello me ayudé de bibliografía como el libro *Sistemas expertos: principios y programación* [6].

Estudié la posibilidad de usar alguna herramienta como *CLIPS* [2], pero decidí que adaptar e incluirlo en mi proyecto iba a ser más costoso que crear un pequeño motor desde cero.

Este motor está compuesto por tres funciones que cumplen las tres reglas básicas para poner grupos en *FreeRummi*: poner escaleras, poner conjuntos y poner fichas mediante los conjuntos que ya están en el tablero.

Con estas tres funciones y las funciones de las clases base, creé tres sistemas expertos de ejemplo contra los que se puede jugar en la **partida simple**

Este incremento también me ocupó bastante tiempo debido a la gran cantidad de pruebas que tuve que realizar con cada una de las funciones del motor.

3.2. Diagrama de Gantt

En primer lugar he incluido los incrementos y las tareas de cada uno:

WBS	Nombre	Inicio	Fin
1	**Requisitos básicos del sistema**	**jul 1**	**ago 4**
1.1	Primera planificación global del proyecto	jul 1	jul 14
1.2	Estudio sobre el juego y sus reglas	jul 15	jul 21
1.3	Toma de requisitos	jul 22	jul 28
1.4	Elección del lenguaje y las herramientas a utilizar	jul 29	ago 4

Figura 3.2: Incremento 1: Selección de herramientas

[2] *CLIPS* es una herramienta que provee un entorno de desarrollo para la producción y ejecución de sistemas expertos. Fue creado a partir de 1984, en el Lyndon B. Johnson Space Center de la NASA.

WBS	Nombre	Inicio	Fin
2	**Organización de las partes básicas del juego**	**ago 5**	**oct 18**
2.1	Estudio de las posibles clases base	ago 5	ago 18
2.2	Análisis de la clase Ficha	ago 19	ago 20
2.3	Implementación de la clase Ficha	ago 23	ago 24
2.4	Prueba de la clase Ficha	ago 25	ago 26
2.5	Análisis de la clase Monton	ago 27	ago 30
2.6	Implementación de la clase Monton	ago 31	sep 1
2.7	Prueba de la clase Monton	sep 2	sep 3
2.8	Análisis de la clase Soporte	sep 6	sep 10
2.9	Implementación de la clase Soporte	sep 13	sep 15
2.10	Prueba de la clase Soporte	sep 16	sep 20
2.11	Análisis de la clase Tablero	sep 21	oct 4
2.12	Implementación de la clase Tablero	oct 5	oct 11
2.13	Prueba de la clase Tablero	oct 12	oct 18
2.14	Clases Base terminadas	oct 15	oct 15

Figura 3.3: Incremento 2: Organización de las partes básicas del juego

WBS	Nombre	Inicio	Fin
3	**Introducción de gráficos y sonidos**	**oct 19**	**nov 23**
3.1	Análisis de la clase Video	oct 19	oct 20
3.2	Implementación de la clase Video	oct 21	oct 22
3.3	Prueba de la clase Video	oct 25	oct 26
3.4	Análisis de las clases Musica y Efecto	oct 27	oct 29
3.5	Implementación de las clases Musica y Efecto	nov 1	nov 3
3.6	Prueba de las clases Música y Efecto	nov 4	nov 5
3.7	Análisis de la clase Fuente	nov 8	nov 9
3.8	Implementación de la clase Fuente	nov 10	nov 11
3.9	Prueba de la clase Fuente	nov 12	nov 15
3.10	Análisis de la clase Iniciado	nov 16	nov 17
3.11	Implementación de la clase Iniciado	nov 18	nov 19
3.12	Prueba de la clase Iniciado	nov 22	nov 23

Figura 3.4: Incremento 3: Introducción de gráficos y sonidos

WBS	Nombre	Inicio	Fin
4	**Diseño e implementación de la clase pártida**	**nov 24**	**feb 15**
4.1	Estudio de las funcionalidades	nov 24	dic 7
4.2	Diseño de las imagenes	nov 24	dic 14
4.3	Elección y edición del audio	nov 24	dic 14
4.4	Análisis de la clase Partida	nov 24	dic 21
4.5	Implementación de la clase Partida	nov 24	ene 4
4.6	Prueba de la clase Partida	ene 5	feb 15
4.7	Modificación de las clases base	nov 24	dic 7
4.8	Análisis de la clase Puntero	dic 8	dic 9
4.9	Implementación de la clase Puntero	dic 10	dic 10
4.10	Prueba de la clase Puntero	dic 13	dic 13
4.11	Análisis de la clase Cronometro	dic 14	dic 15
4.12	Implementación de la clase Cronometro	dic 16	dic 17
4.13	Prueba de la clase Cronometro	dic 20	dic 21
4.14	Análisis de la clase Jugador	dic 22	dic 31
4.15	Implementación de la clase Jugador	ene 3	ene 7
4.16	Prueba de la clase Jugador	ene 10	ene 14
4.17	Partida Finalizada	ene 14	ene 14

Figura 3.5: Incremento 4: Diseño e implementación de la clase Partida

WBS	Nombre	Inicio	Fin
5	**Diseño e implementación de los menús**	**feb 16**	**mar 1**
5.1	Diseño de imagenes de Menu	feb 16	mar 1
5.2	Análisis de la clase Menu	feb 16	feb 22
5.3	Implementación de la clase Menu	feb 23	feb 25
5.4	Prueba de la clase Menu	feb 28	mar 1

Figura 3.6: Incremento 5: Diseño e implementación de menús

WBS	Nombre	Inicio	Fin
6	**Diseño e implementación del gestor de partida**	**mar 2**	**mar 21**
6.1	Análisis de la clase GestorPartida	mar 2	mar 11
6.2	Implementación de la clase GestorPartida	mar 14	mar 17
6.3	Prueba de la GestorPartida	mar 18	mar 21

Figura 3.7: Incremento 6: Diseño e implementación del gestor de partidas

WBS	Nombre	Inicio	Fin
7	**Diseño e implementación del gestor de jugadores**	**mar 22**	**abr 4**
7.1	Análisis de la clase GestorJugadores	mar 22	mar 28
7.2	Implementación de la clase GestorJugadores	mar 29	mar 31
7.3	Prueba de la clase GestorJugadores	abr 1	abr 4

Figura 3.8: Incremento 7: Diseño e implementación del gestor de jugadores

WBS	Nombre	Inicio	Fin
8	**Diseño e implementación de los sistemas expertos**	**abr 5**	**may 5**
8.1	Estudio de posibles herramientas para sistemas experto:	abr 5	abr 8
8.2	Análisis de la clase IA	abr 11	abr 20
8.3	Implementación de la clase IA	abr 21	abr 28
8.4	Prueba de la clase IA	abr 29	may 5

Figura 3.9: Incremento 8: Diseño e implementación de los sistemas expertos

WBS	Nombre	Inicio	Fin
9	Realización de la memoria	jul 1	ago 25
10	Entrega del producto	may 2	may 1

Figura 3.10: Otras tareas

A continuación se muestra el diagrama de Gantt, con la planificación que he ido siguiendo para realizar el proyecto:

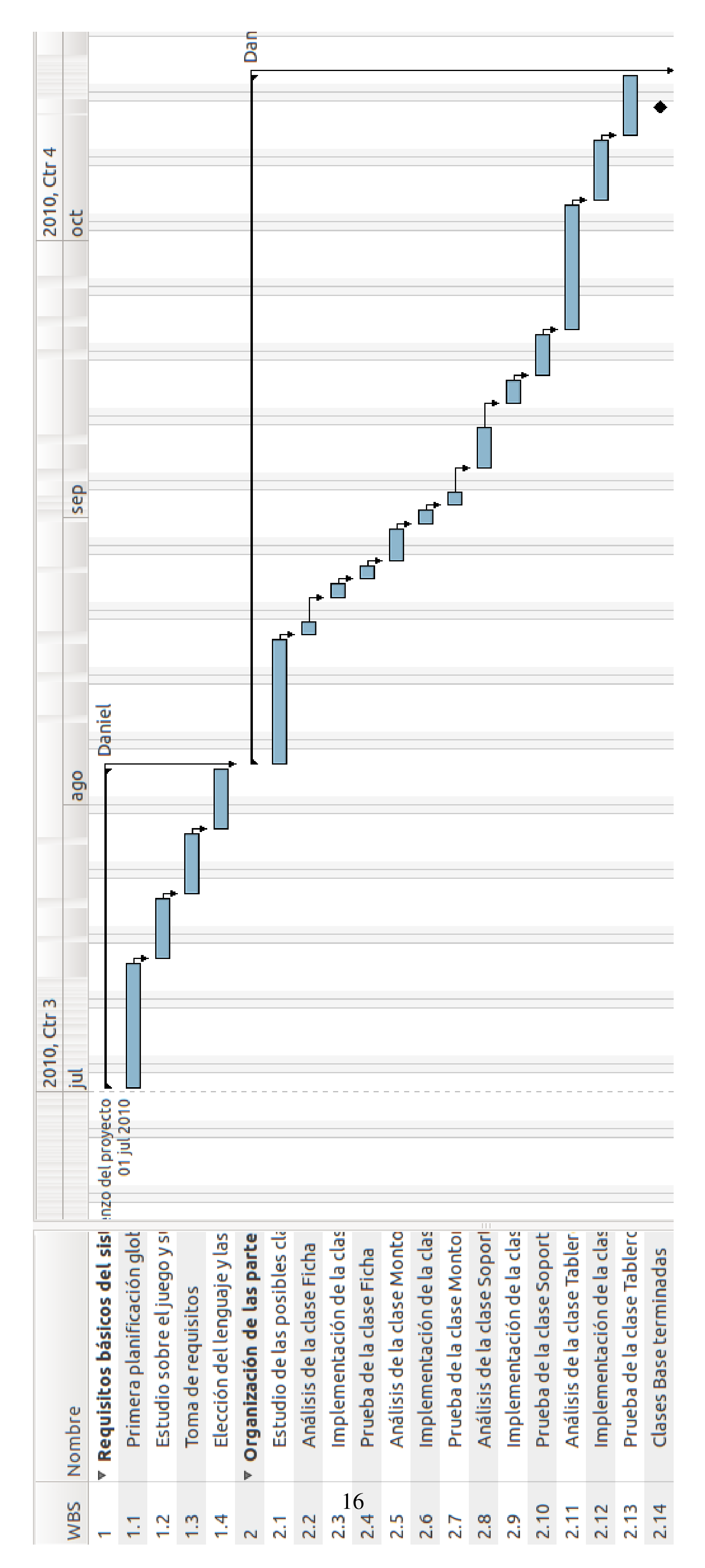


Figura 3.11: Diagrama de Gantt. Incrementos 1 y 2

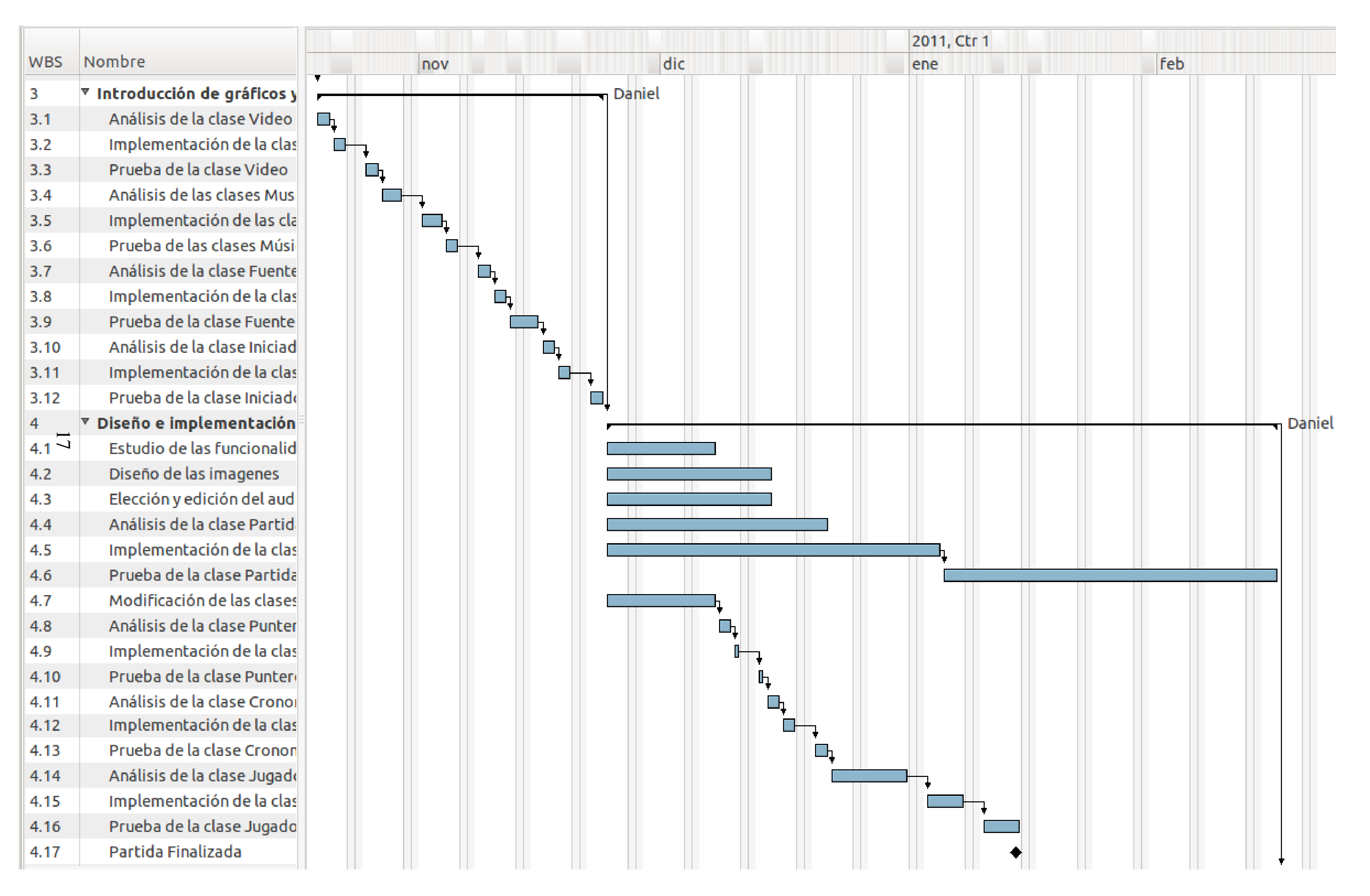


Figura 3.12: Diagrama de Gantt. Incrementos 3 y 4

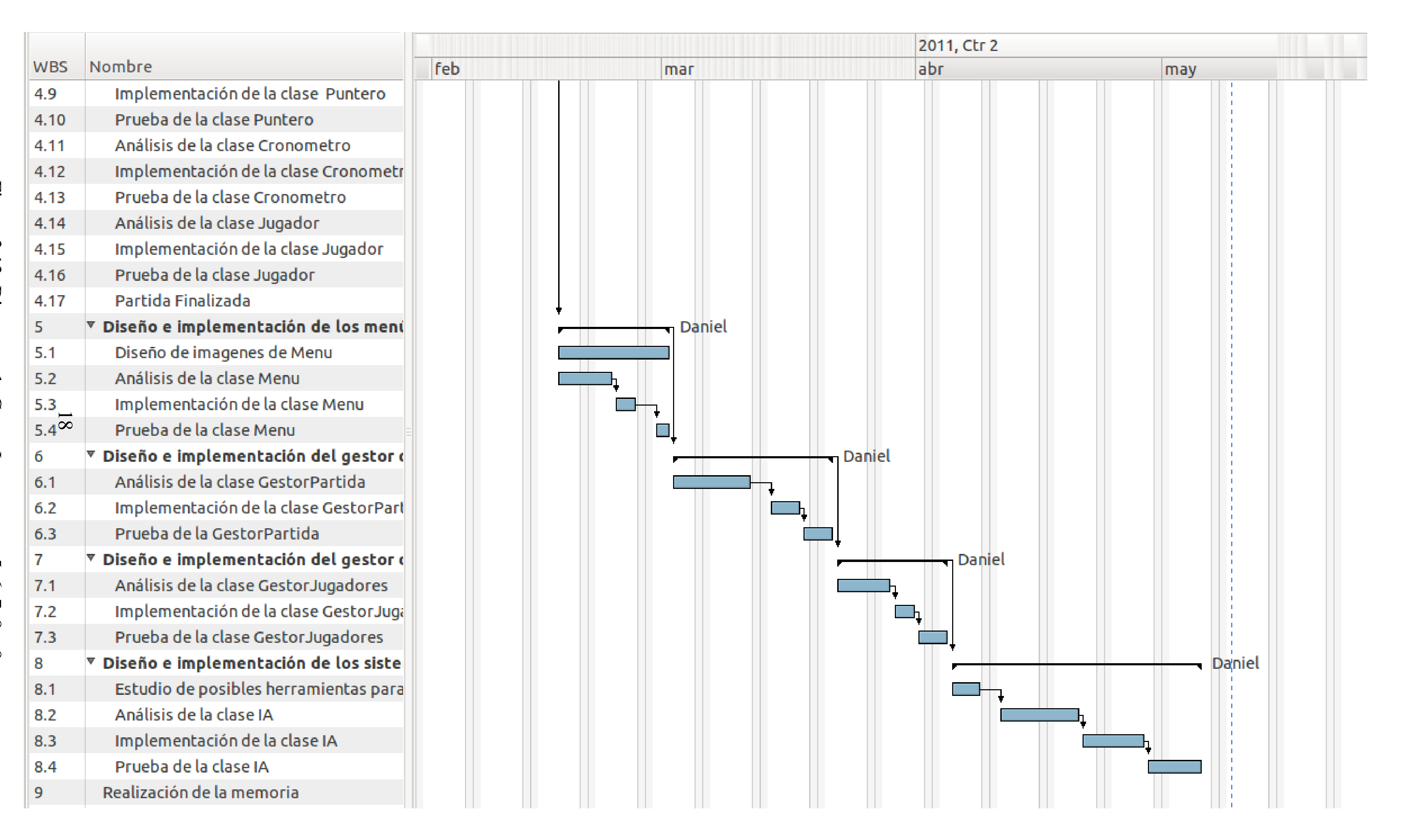


Figura 3.13: Diagrama de Gantt. Incrementos 5, 6, 7, 8 y 9

Capítulo 4

Especificación de los requisitos del sistema

4.1. Modelado

En este capitulo y en los dos siguientes voy a describir el modelado de la aplicación. Dicho modelado comprende el Análisis y el Diseño del software que debemos realizar antes de la codificación.

El modelado nos proporciona la documentación necesaria para desarrollar, implementar y mantener el proyecto, así como una base para diseñar los planes de prueba del software.
Para el modelado de este sistema, dadas sus características, vamos a usar el modelo orientado a objetos usando la notación UML[1].

En este capítulo trataré primeramente la especificación de los requisitos del sistema, para ello he creado una lista lo más detallada y completa posible donde se reúnan todos los requisitos que debe cumplir el software.

4.2. Requisitos de interfaces externas

En este apartado se van a describir los requisitos de conexión entre el software y el hardware, así como la interfaz del usuario.

De la interfaz entre el software y el hardware se encarga la librería SDL. Al ser un sistema preestablecido, no tendré que analizarlo ni diseñarlo, solo haré uso de él.

Pasamos a definir el interfaz entre el videojuego y el usuario. Todas las ventanas de la aplicación podrán ser mostradas a pantalla completa o en formato de ventana con una resolución de 1024x768 píxeles. A continuación se definen las distintas ventanas con las que el usuario se puede encontrar:

- **Ventana Menú principal**: Esta ventana mostrará el menú principal de *FreeRummi* y mostrará el nombre de la aplicación, así como las opciones para **Jugar**, para editar los **Jugadores** y para **Salir**. El menú será completamente manejado por el ratón y bastará un clic encima de una opción para acceder a ella. Por último habrá una opción para silenciar el juego que también podrá ser manejada con el ratón.

- **Ventana de Menú Jugar**: Esta ventana mostrará el submenú **Jugar** y mostrará las opciones para jugar una **partida simple**, jugar un **Concurso** y para **Volver** al menú principal. El menú será

[1] UML es un lenguaje de especificación para definir un sistema software que detalla la funcionalidad del sistema y que documenta el proceso de desarrollo del mismo.

completamente manejado por el ratón y bastará un clic encima de una opción para acceder a ella. Por último habrá una opción para silenciar el juego que también podrá ser manejada con el ratón.

- **Ventana del Menú Jugadores**: Esta ventana mostrará el submenú **Jugadores** y mostrará las opciones para **Ver** las estadísticas de un jugador, **Borrar** jugadores y para **Volver** al menú principal. El menú será completamente manejado por el ratón y bastará un clic encima de una opción para acceder a ella. Por último habrá una opción para silenciar el juego que también podrá ser manejada con el ratón.
- **Ventana de Partida**: Esta ventana proporciona la interfaz para interactuar en una partida de *FreeRummi*. Mostrará el tablero, el montón, los soportes, el cronómetro y la información de los jugadores. Como en todos los casos anteriores será completamente manejable por ratón y pulsando la tecla ESC se volverá al menú principal. También poseerá la opción de silenciar el juego.
- **Ventana Ver**: Esta ventana proporciona la interfaz para visualizar los datos estadísticos de un jugador. Es una pantalla meramente informativa. Mediante el ratón se podrá desplazar por los diferentes usuarios que existan en el sistema con ayuda de unos botones. Poseerá un botón para volver al menú principal y otro para silenciar el juego.
- **Ventana Borrar**: Esta ventana proporciona la interfaz para borrar un jugador. Mediante el ratón se podrá desplazar por los diferentes usuarios que existan en el sistema con ayuda de unos botones. Poseerá un botón para confirmar el borrado y otro para cancelar y volver al menú principal así como otro para silenciar el juego.

4.3. Requisitos funcionales

Estos son, a grandes rasgos, los requisitos funcionales que el sistema debe permitir al usuario:

- Jugar una partida a *FreeRummi* tanto a un solo jugador como un grupo.
- Jugar contra el ordenador o contra otras personas.
- Poder pausar la partida.
- Ver las estadísticas de cualquier jugador.
- Poder borrar un jugador del sistema.
- Poder silenciar el juego en cualquier momento.
- Poder salir del juego en cualquier momento.

En estos requisitos no están incluidos algunos específicos de la partida, ya que se irán incorporando a medida que se vaya desarrollando el software.

4.4. Requisitos de rendimiento

Al tratarse un juego que se puede considerar a tiempo real, hay que hacer incapié en que la respuesta al usuario sea lo más rápida posible, sacrificando si es necesario el consumo de memoria principal.

Otro requisito a tener en cuenta es la carga de CPU, intentando optimizar este apartado para que el juego vaya fluido sin consumir en exceso.

4.5. Restricciones de diseño

En el diseño de la aplicación tienen que primar los tiempos de respuesta sobre el consumo de recursos de espacio como la memoria principal o secundaria. Esta es la principal restricción que tendrá el diseño de nuestro videojuego.

Los videojuegos están pensados para ejecutarse como aplicación principal, no para compartir recursos con otros programas,es por ello que pueden consumir muchos recursos.

4.6. Atributos del sistema software

Los atributos que se deben cumplir son:

- La aplicación tiene que ser portable en sistemas compatibles con C++ y SDL.
- El código de la aplicación no debe ser dependiente del sistema operativo en el que se desarrolle la aplicación.
- Debe ser un código mantenible y fácilmente ampliable para futuras mejoras y versiones.

Capítulo 5

Análisis

5.1. Análisis del sistema

Como se dijo en el capítulo anterior, de entre todas las opciones, para este proyecto voy a usar el enfoque orientado a objetos usando la notación UML.

5.2. Funcionalidades del sistema

5.2.1. Organización de las partes básicas del juego

En este incremento se analizan los elementos básicos del juego: ficha, montón, soporte y tablero.

Análisis de la Ficha

La **Ficha** es el elemento más básico para poder jugar. En el juego habrá un total de 106 fichas: 104 fichas numeradas del 1 al 13 en 4 colores: verde, azul, rojo y negro, existiendo dos fichas con el mismo número y color; y 2 comodines.

Análisis del Montón

El **Montón** es el conjunto de todas las fichas del juego. En principio estarán las 106 fichas dentro de él y a medida que se vayan necesitando se irán extrayendo. El Montón debe simular una bolsa o saco en el que se meten las fichas y se remueve para desordenarlas, por tanto se puede decir que en el montón las fichas estarán desordenadas.

Análisis del Soporte

El **Soporte** es el lugar donde el usuario almacena sus fichas, con las cuales tiene que formar combinaciones para deshacerse de ellas. Al principio de una partida se insertarán 14 fichas en él y a medida que vaya formando combinaciones o pueda poner fichas en el tablero las irá extrayendo. Si en un momento dado no hay fichas que poner, se extraerá una ficha del Montón y se insertará en el soporte.

Análisis del Tablero

El **Tablero** es el lugar donde los usuarios ponen sus combinaciones. el sistema debe comprobar que las combinaciones son correctas, en caso contrario, las combinaciones volverán a su soporte. Las fichas que estén colocadas en el tablero podrán ser manipuladas por todos los usuarios.

5.2.2. Introducción de gráficos y sonidos

Uno de los aspectos fundamentales de un videojuego es la interfaz gráfica. La aplicación dispone de una interfaz agradable y sencilla, pudiendo ser manejada completamente con el ratón. Esto hace que la forma de seleccionar una opción, de realizar acciones durante la partida, etc, se haga con una sola pulsación. Además los contenidos están bien organizados y con posible retorno en los menús con la opción volver. Aunque la interfaz es sencilla, creo que esto ayuda a que todo tipo de jugadores puedan acceder al juego de forma fácil, sin tener demasiada soltura en el manejo de un dispositivo informático.

Los gráficos son creación mía en su totalidad, se podría mejorar este aspecto del juego, pero debido a que soy yo el que tiene que encargarse de todo el proyecto, no he podido emplear las horas que este campo necesita.

En cuanto al aspecto sonoro, los efectos están extraídos del *banco de imágenes y sonidos del instituto de tecnologías educativas (ITE)* [4]. Mientras que la banda sonora, tanto de los menús como de la partida están extraídos del sitio web *Jamendo* [7]. Posteriormente retoqué las canciones, quedándome solo con el loop de música que me interesaba.

5.2.3. La clase Partida

Una vez explicadas las partes básicas del juego, así como los elementos gráficos y sonoros, ya se puede hablar de una partida completa.

A los elementos que se explicaron en el incremento dos, hay que sumar ahora elementos como los jugadores y el cronómetro.

En el incremento dos solo se podía jugar en consola y con unas funcionalidades muy limitadas. Una partida real de *FreeRummi*, tiene varios jugadores, cada uno con sus puntuaciones, su nombre y sus estadísticas. Además los jugadores hay que crearlos o cargarlos en memoria, guardar sus puntaciones una vez han finalizado la partida y borrarlos en el caso de que ya no se quieran usar más.

Por otro lado el cronómetro es que impondrá orden en los turnos de los jugadores, limitando estos a 60 segundos de duración. El cronómetro se puede pausar y reanudar en un momento dado, imposibilitando al jugador realizar movimientos mientras esté en pausa.

En cuanto a la Partida en sí, el juego se compone de dos tipos de partida: **Simple** y **Concurso**.

La **partida simple**, es una partida de entrenamiento en la que no cuenta la puntuación que se obtenga, solo sirve para adquirir experiencia.

El **concurso**, se compone de tantas partidas como jugadores haya, y se tiene en cuenta la puntuación.

5.2.4. Los menús, el gestor de partida y el gestor de jugadores

Cuando ya tuve la partida finalizada, empecé a añadir los menús para poder navegar por la aplicación.

Aunque en este incremento creé el menú principal, en los siguientes incrementos tuve que ir modificando y añadiendo nuevas opciones según iba añadiendo funcionalidades al sistema.

Por otro lado el gestor de partida trata de facilitar al usuario la forma de seleccionar el número de jugadores que participarán en una partida y qué jugadores participarán.

Por último, para darle una sensación de continuidad y de reto, se crea el gestor de jugadores. Esto nos permite crearnos un jugador y jugar con el siempre que utilicemos el juego. Con esto conseguiremos estadísticas de nuestras partidas y nuevos retos a conseguir.

La gestión de jugadores incluye: ver estadísticas y borrar jugadores.

En los siguientes apartados, se hace un estudio más a fondo de estos elementos del juego.

5.2.5. Sistemas expertos

Este apartado se trata de un añadido a la clase partida, en particular a la partida simple: se cambio la forma de jugar la partida simple, en vez de jugarse contra otros jugadores, se juega contra el ordenador.

La forma de actuar del sistema experto variará en función de la inteligencia que implemente. El sistema experto deberá poder poner conjuntos, escaleras y fichas sueltas, así como una combinación de estas tres cosas.

5.3. Modelo de casos de uso

El **modelo de casos de uso de UML** especifica que comportamiento debe tener el sistema. representa los requisitos funcionales del sistema centrándose en qué hace y no en cómo lo hace.

5.3.1. Diagrama de casos de uso

En este apartado se incluye el diagrama de casos de uso que representa la funcionalidad de *FreeRummi*. Para ello he seguido el siguiente esquema:

- Identificación de los usuarios del sistema y sus roles.
- Para cada role, identificación de todas las maneras de interactuar con el sistema.
- Creación de casos de uso para cada objetivo que queramos cumplir.
- Estructuración de dichos casos de uso.

El resultado de aplicar este esquema a mi proyecto es el que se puede visualizar en la figura 5.1:

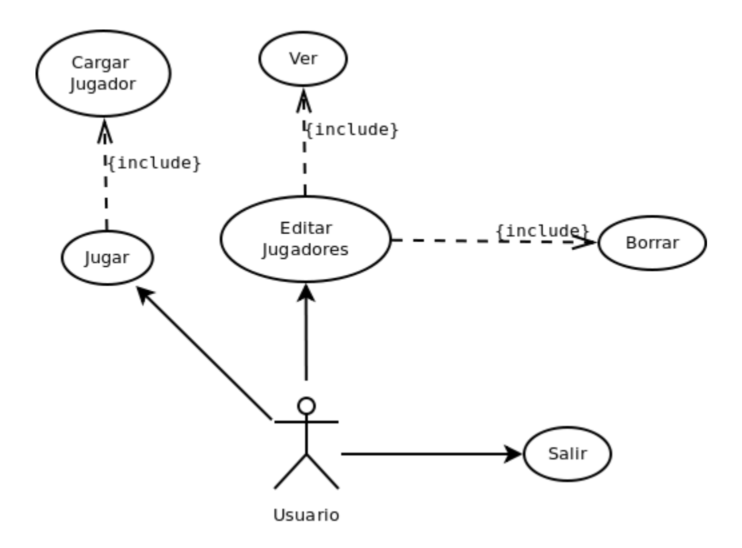


Figura 5.1: Diagrama de casos de uso del sistema

5.3.2. Descripción de los casos de uso

Para la descripción de los casos de uso se va a utilizar una notación formal usando plantillas. Este texto debe ser legible y comprensible por un usuario que no sea experto.

Descripción del caso de uso: Cargar jugador

Caso de uso: Cargar Jugador
Descripción: Carga un jugador para jugar una partida a *FreeRummi*.
Actores: Usuario.
Precondiciones: -
Postcondiciones: Se carga un jugador para jugar una partida.
Escenario principal:

1. El *Usuario* desea cargar un jugador.
2. El *Sistema* muestra el menú para escribir el nombre del jugador.
3. El *Usuario* introduce el nombre del jugador.
4. El *Sistema* comprueba que el nombre introducido es válido y que existe el jugador.
5. El *Sistema* carga el jugador.

Escenario alternativo:

*a. En cualquier momento un usuario puede cancelar y salir.
    El *Sistema* cierra el menú y sale al menú principal.

4a. El nombre introducido no es válido.
    4a1. El *Sistema* muestra un mensaje por pantalla y vuelve al paso *2*.
4b. El jugador no existe en el sistema.
    4b1. El *Sistema* crea un jugador con el nombre introducido.

Descripción del caso de uso: Ver

Caso de uso: Ver
Descripción: Muestra las estadísticas de los usuarios.
Actores: Usuario.
Precondiciones: Deben existir al menos un jugador en el sistema.
Postcondiciones: Un jugador será mostrado por pantalla.
Escenario principal:

1. El *Usuario* desea ver las estadísticas de los jugadores.
2. El *Sistema* muestra un menú para seleccionar el jugador que se desea visualizar.
3. El *Usuario* elige el jugador que desea visualizar.
4. El *Sistema* muestra por pantalla las estadísticas del jugador seleccionado.

Escenarios alternativos:

*a. En cualquier momento el usuario puede salir.

2a. No existe ningún jugador en el sistema.
    2a1. El *Sistema* muestra por pantalla un mensaje diciendo que no existe ningún jugador.

4a. El jugador seleccionado no existe.
    4a1. El *Sistema* muestra un mensaje por pantalla y vuelve al paso *2*.

Descripción del caso de uso: Borrar

Caso de uso: Borrar
Descripción: Borra a un jugador del sistema.
Actores: Usuario.
Precondiciones: Deben existir al menos un jugador en el sistema.
Postcondiciones: Un jugador será borrado del sistema.
Escenario principal:

1. El *Usuario* desea borrar algún jugador.
2. El *Sistema* muestra un menú para seleccionar el jugador que se desea borrar.
3. El *Usuario* elige el jugador que desea borrar.
4. El *Sistema* borra al jugador seleccionado.

Escenarios alternativos:

*a. En cualquier momento el usuario puede salir.
2a. No existe ningún jugador en el sistema.
    2a1. El *Sistema* muestra por pantalla un mensaje diciendo que no existe ningún jugador.
4a. El jugador seleccionado no existe.
    4a1. El *Sistema* muestra un mensaje por pantalla y vuelve al paso *2*.

Descripción del caso de uso: Jugar

Caso de uso: Jugar
Descripción: El jugador juega una partida al videojuego.
Actores: Usuario.
Precondiciones: Existen jugadores en el sistema.
Postcondiciones: Se jugará una partida al videojuego.
Escenario principal:

1. El *Usuario* desea jugar una partida.
2. El *Sistema* muestra un menú para seleccionar el número de jugadores.
3. El *Usuario* selecciona el número de jugadores que participarán en la partida.
4. El *Sistema* comprueba que el número de jugadores es correcto.
5. **include** Cargar Jugador
6. El *Sistema* comprueba que el jugador no está ya reservado y vuelve al paso *5* tantas veces como jugadores haya seleccionado.
7. El *Sistema* inicializa y muestra la partida por pantalla.
8. El *Usuario* y el *Sistema* interactúan durante la partida.
9. El *Sistema* muestra quien ha ganado al final de la partida.
10. El *Sistema* actualiza y guarda las nuevas puntuaciones de los jugadores.
11. El *Sistema* cierra la partida y muestra el menú principal.

Escenarios alternativos:

*a. En cualquier momento el usuario puede cancelar el proceso.
El *Sistema* cierra la partida y vuelve al menú principal.

4a. El número de jugadores no es válido.
4a1. El *Sistema* vuelve al paso *2*.

6a. El jugador seleccionado ya está reservado para la partida.
6a1. El *Sistema* muestra un mensaje por pantalla y vuelve al paso *5*.

Descripción del caso de uso: Editar Jugadores

Caso de uso: Editar Jugadores
Descripción: Gestión de los jugadores del juego.
Actores: Usuario.
Precondiciones: Debe existir al menos un jugador en el sistema.
Postcondiciones: El usuario podrá ver las estadísticas o borrar a un jugador.
Escenario principal:

1. El *Usuario* desea gestionar algún jugador.

2. El *Sistema* muestra un menú con las opciones de gestión.
3. El *Usuario* elige una opción.
4. El *Sistema* realiza la acción según la opción.

Escenarios alternativos:
*a. En cualquier momento el usuario puede cancelar el proceso.
El *Sistema* cierra el menú y vuelve al menú principal.

3a. El *Usuario* decide ver las estadísticas de un jugador.
3a1. **include** Ver.

3b. El *Usuario* decide borrar a un jugador del sistema.
3b1. **include** Borrar.

5.4. Modelo conceptual de datos

Este apartado del análisis sirve para especificar los requisitos del sistema y las relaciones estáticas que existen entre ellos.

Para este fin se utiliza como herramienta los diagramas de clase. En estos diagramas se representan las clases de objetos, las asociaciones entre dichas clases, los atributos que componen las clases y las relaciones de integridad.

5.4.1. Descripción diagramas de clases conceptuales

En este apartado se presenta una lista de las clases que formarán parte de mi sistema y una pequeña descripción de qué hace cada una.

- **Cronometro:** Clase para el manejo y funcionamiento del cronómetro del juego.
- **Ficha:** Clase para el manejo y funcionamiento de las fichas del juego.
- **Fuente:** Clase para gestionar el trabajo con la librería *ttf*.
- **GestorJugadores:** Clase para gestionar los menús referentes a los jugadores.
- **GestorPartida:** Clase para gestionar los menús referentes a la partida.
- **IA:** Clase para gestionar la inteligencia artificial y los sistemas expertos del juego.
- **Imagen:** Clase para gestionar el trabajo con imágenes.
- **Iniciado:** Clase para arrancar las librerías de SDL.
- **Jugador:** Clase para el manejo y la gestión de los jugadores.
- **Menu:** Clase para gestionar el menú principal y los submenús.
- **Monton:** Clase para el manejo y funcionamiento del montón del juego.
- **Partida:** Clase para el manejo y funcionamiento de la partida de FreeRummi.
- **Puntero:** Clase para el manejo y funcionamiento del puntero del ratón.

- **Sonido:** Clase para gestionar el trabajo con la librería *mixer*.
- **Soporte:** Clase para el manejo y funcionamiento del soporte de fichas del juego.
- **Tablero:** Clase para el manejo y funcionamiento del tablero del juego.
- **Video:** Clase para gestionar el sistema de vídeo.

5.4.2. Diagrama de clases

Para descomponer el diagrama he tomado el criterio de funcionalidad, para ello se mostrarán subconjuntos de clases que tengan un fin común.

Debido al gran tamaño del diagrama resultante, he optado por hacerlo por partes y luego unir esas partes en forma de caja negra para formar el diagrama final mucho más fácil de comprender.

El primero de estos diagramas de clases es el referente a la **Partida**, el más complejo, debido a que será el pilar central del juego. Puede observarse en la figura 5.2:

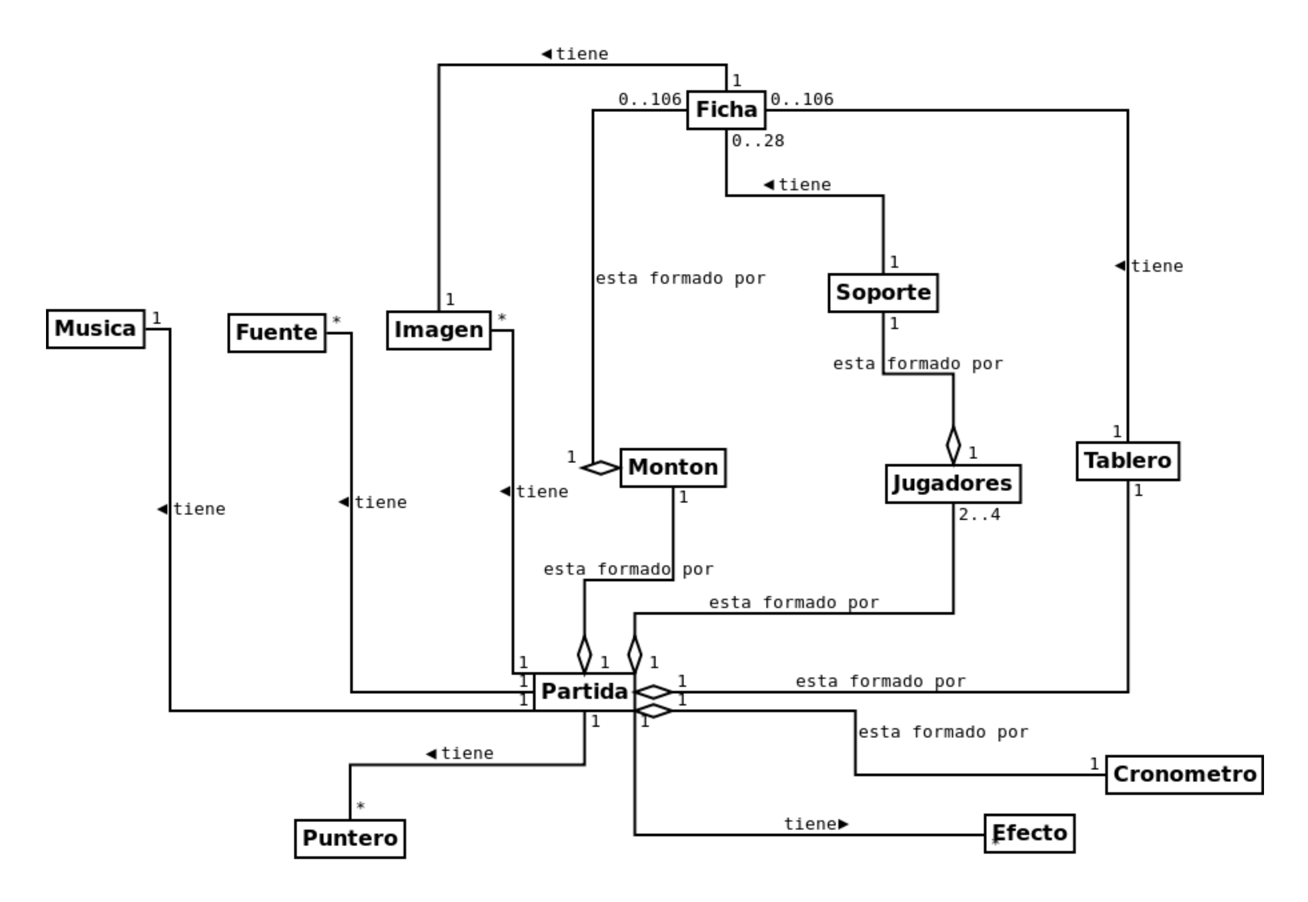


Figura 5.2: Diagrama de clases: Partida

El segundo diagrama que se presenta, relaciona las clases para poder gestionar a los jugadores 5.3:

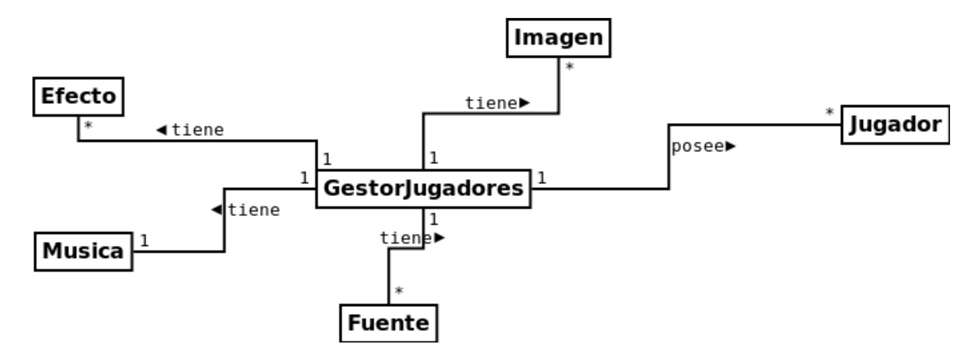


Figura 5.3: Diagrama de clases: Gestor de jugadores

El tercer y último diagrama parcial, relaciona las clases para llevar a cabo el gestor de la partida 5.4:

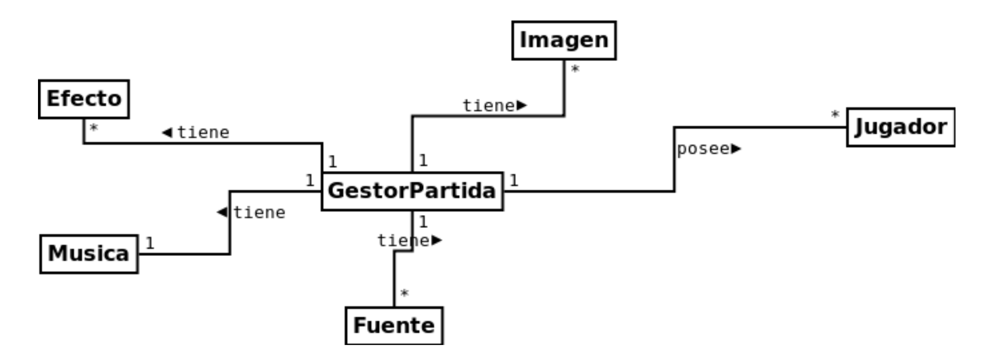


Figura 5.4: Diagrama de clases: El gestor de partida

Y para terminar este apartado se presenta el esquema que muestra todo lo expuesto anteriormente y sus relaciones 5.5:

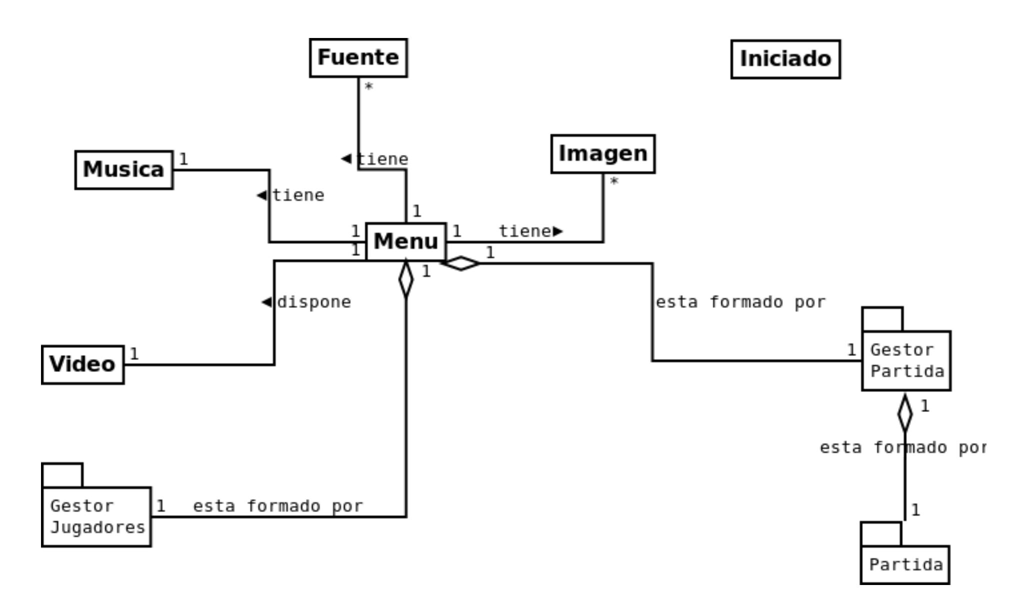


Figura 5.5: Diagrama de clases: Diagrama Global

5.5. Modelo de comportamiento del sistema

El **modelo de comportamiento** especifica cómo debe de actuar un sistema. El sistema a considerar es el que engloba a todos los objetos. Este modelo consta de dos partes:

- El **Diagrama de secuencias del sistema** el cual muestra la secuencia de eventos entre los actores y el sistema.

- Los **Contrato de las operaciones del sistema** que describen el efecto que producen las operaciones del sistema.

5.5.1. Diagramas de secuencias del sistema y Contrato de las operaciones del sistema

Diagrama de secuencia: Cargar Jugador

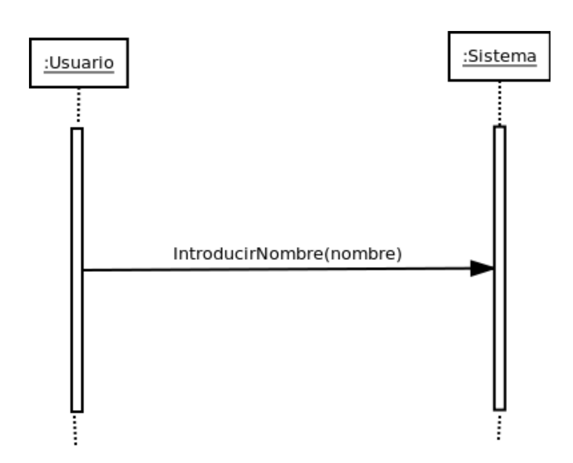


Figura 5.6: Diagrama de secuencia: Cargar Jugador

Contratos de operaciones: Cargar Jugador

Operación: IntroducirNombre(nombre)
Responsabilidades: Cargar un jugador en memoria.
Referencias cruzadas: Caso de uso: Jugar.
Precondiciones: Debe existir un fichero con el nombre de todos los jugadores del sistema.
Postcondiciones:

- Se **creó** una instancia **J** de **Jugador**.

- Si el nombre del jugador estaba en el fichero de con todos los nombres:

  - Se cargó en memoria el fichero con los datos de dicho jugador y se **asignaron** los datos a **J** (modificación de atributos).

- Si el nombre del jugador NO estaba en el fichero:

  - Se **creó** un fichero con los datos del nuevo jugador y se **asignaron** a **J** (modificación de atributos).

Diagrama de secuencia: Ver

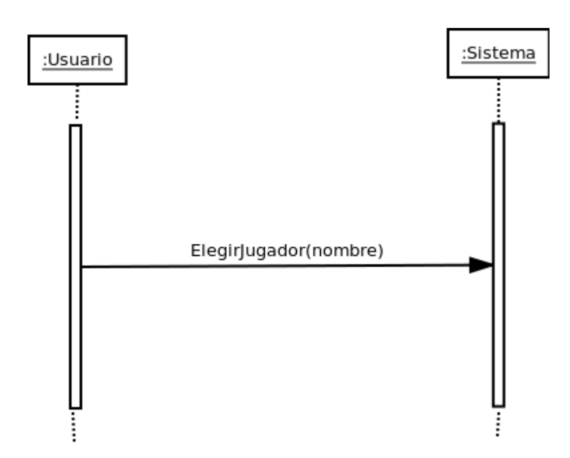


Figura 5.7: Diagrama de secuencia: Ver Jugador

Contratos de operaciones: Ver

Operación: ElegirJugador(nombre)
Responsabilidades: Mostrar las estadísticas de un jugador por pantalla.
Referencias cruzadas: Caso de uso: Editar Jugadores.
Precondiciones: Debe existir un fichero con el nombre de todos los jugadores del sistema y debe existir en él una entrada que coincida con **nombre**
Postcondiciones:

- Se **mostró** por pantalla la información correspondiente al jugador con **nombre** = **J.nombre**.

Diagrama de secuencia: Borrar

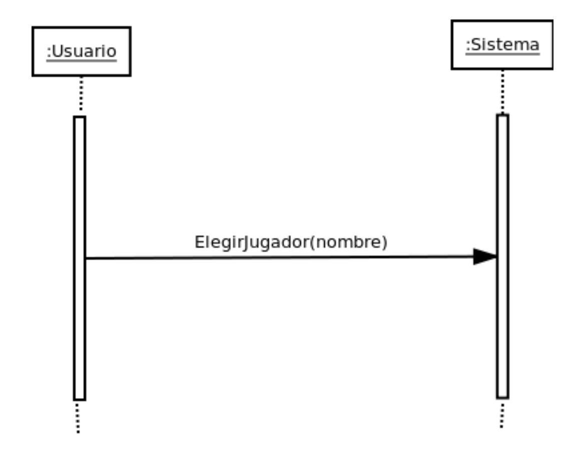


Figura 5.8: Diagrama de secuencia: Borrar Jugador

Contratos de operaciones: Borrar

Operación: ElegirJugador(nombre)
Responsabilidades: Borra a un jugador del sistema.

Referencias cruzadas: Caso de uso: Editar Jugadores.
Precondiciones: Debe existir un fichero con el nombre de todos los jugadores del sistema y debe existir en él una entrada que coincida con **nombre**
Postcondiciones:

- Se **borró** el fichero con la información referente al jugador cuyo atributo nombre coincidía con **nombre**

Diagrama de secuencia: Jugar

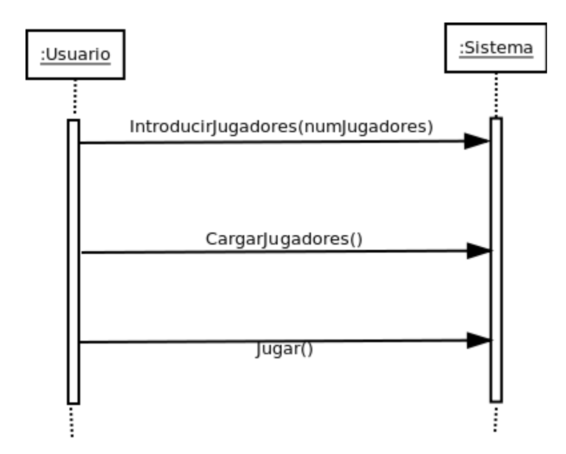


Figura 5.9: Diagrama de secuencia: Jugar Partida

Contratos de operaciones: Jugar partida

Operación: IntroducirJugadores(numJugadores)
Responsabilidades: Confirmar el número de jugadores que participarán en la partida.
Precondiciones: Debe ser un número comprendido entre dos y cuatro
Postcondiciones:

- Se **creó** una instancia **P** de **Partida**.
- Se **asignó numJugadores** a **P.nomJugadores**.

Operación: CargarJugadores()
Responsabilidades: Carga en la partida tantos jugadores como se haya indicado en la operación anterior.
Precondiciones: -
Postcondiciones:

- Se **crearon** tantas instancias **Jx** de **Jugador**, como **P.numJugadores** haya.
- Se **asignaron** los jugadores elegidos a **Jx**.

Operación: Jugar()
Responsabilidades: Jugar una partida a FreeRummi.
Precondiciones: Los jugadores deben estar correctamente cargados.
Postcondiciones:

- Se llevó a cabo una partida a FreeRummi.
- Se **asignaron** los valores del resultado de la partida a los Jugadores **Jx**.

Diagrama de secuencia: Editar Jugadores

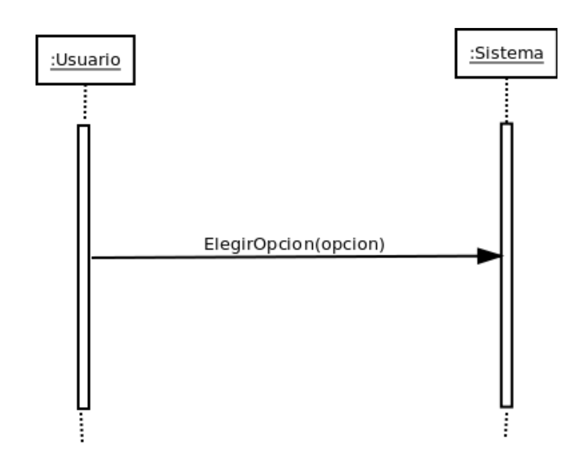


Figura 5.10: Diagrama de secuencia: Editar Jugador

Contratos de operaciones: Editar Jugadores

Operación: ElegirOpcion(opcion)
Responsabilidades: Muestra por pantalla un menú con las opciones de edición.
Preconditiones: -
Postcondiciones:

- Se accedió al submenú correspondiente a **opcion**.

Capítulo 6

Diseño

6.1. Diseño del sistema

Al igual que se hizo en el capitulo de análisis, para el tema del diseño también se seguirá una metodología orientada a objetos mediante UML.

Este proceso es mucho más sencillo una vez que hemos especificado qué hace el sistema en los capítulos anteriores. Cabe decir que al igual que en el análisis, el diseño del sistema, no contempla muchos detalles del sistema final, solo da una idea orientativa sobré como se implementará el sistema.

En este capítulo no he incluido descripciones de las clases ni de los métodos o atributos de las mismas, dado que en la documentación del código se encuentra todo lo necesario a cerca de las clases y archivos que componen la aplicación.

6.2. Diagrama de clases de diseño

A continuación se presenta el diagrama de diseño de *FreeRummi* 6.1:

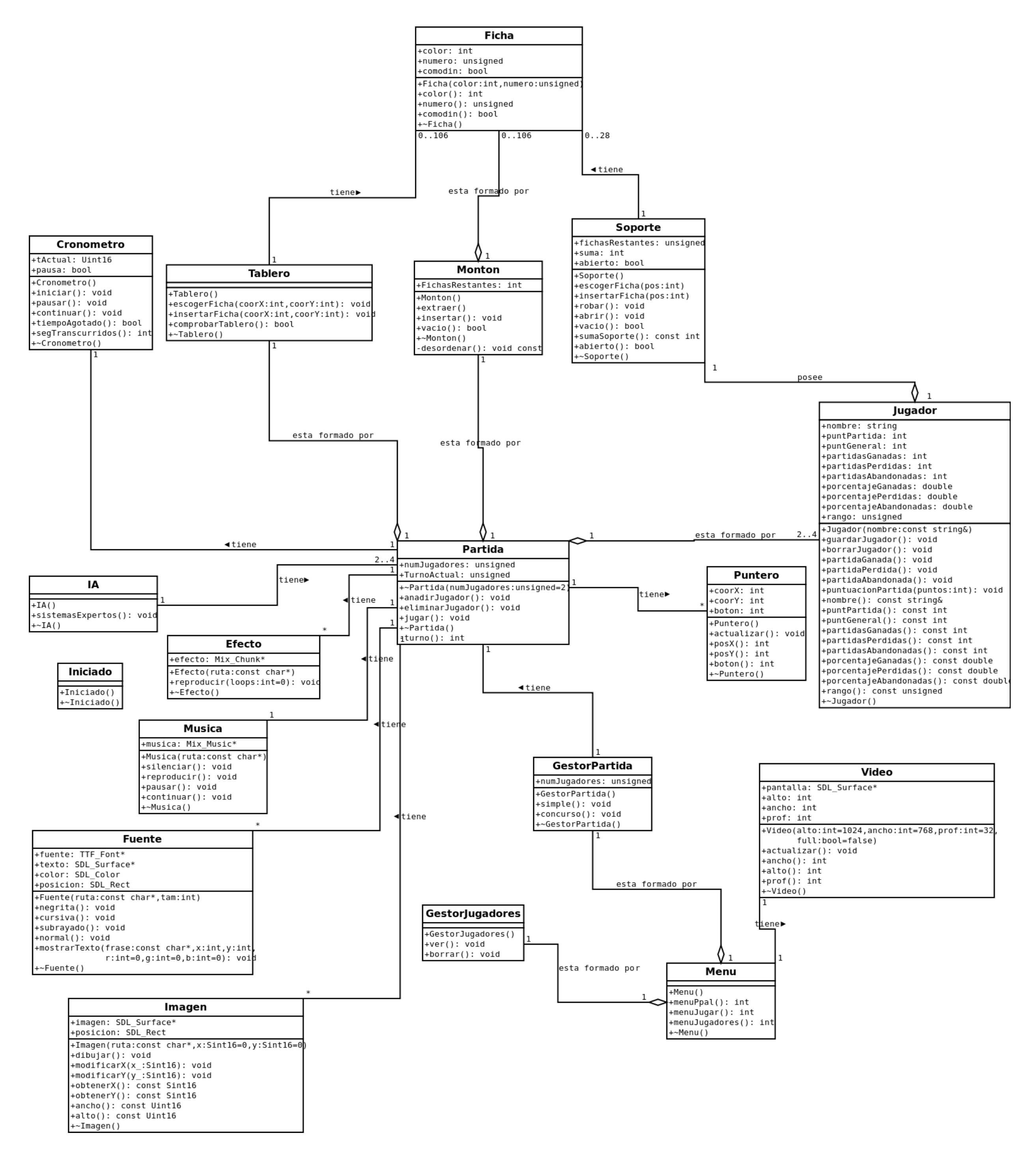


Figura 6.1: Diagrama de diseño del sistema

6.3. Organización de las partes básicas del juego

En este apartado repasaré el diseño que he realizado de las partes base del juego.

6.3.1. Clase Ficha

Como se comentó en el análisis la ficha debe componerse por un número y un color, para el número se usará un natural, ya que el número está comprendido entre 1 y 13 (0 si es una ficha nula), y el color será un enumerado.

La clase deberá tener métodos para construir fichas, copiarlas, asignarlas, así como para obtener el número y el color.

6.3.2. Clase Monton

El montón será un vector de fichas. También guardaremos como atributo el número de fichas restantes en dicho vector.

Esta clase deberá permitir construir un montón, desordenarlo y poder extraer fichas de él.

6.3.3. Clase Soporte

El soporte será también un vector de fichas. Tendrá otros atributos como la puntuación (que será la suma de los valores numéricos de las fichas) y el número de fichas restantes.

La clase permitirá construir un soporte, copiarlo, insertar y extraer fichas de él, así como obtener información sobre la puntuación y las fichas restantes.

6.3.4. Clase Tablero

El tablero será una matriz de fichas.

La clase dará la opción de construir y copiar un soporte, de insertar fichas en él, así como comprobar que las combinaciones que se pongan son correctas.

6.4. La clase Partida

6.4.1. Diagramas de secuencia de los casos de uso

Diagrama de secuencia del caso de uso CargarJugador

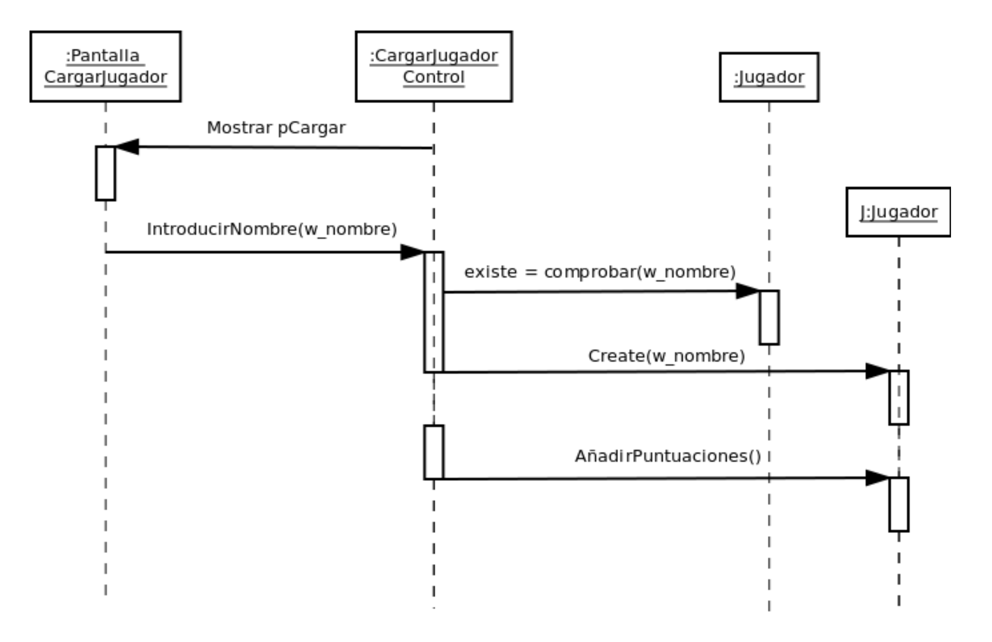


Figura 6.2: Diagrama de secuencia del caso de uso Cargar Jugador

Diagrama de secuencia del caso de uso Jugar

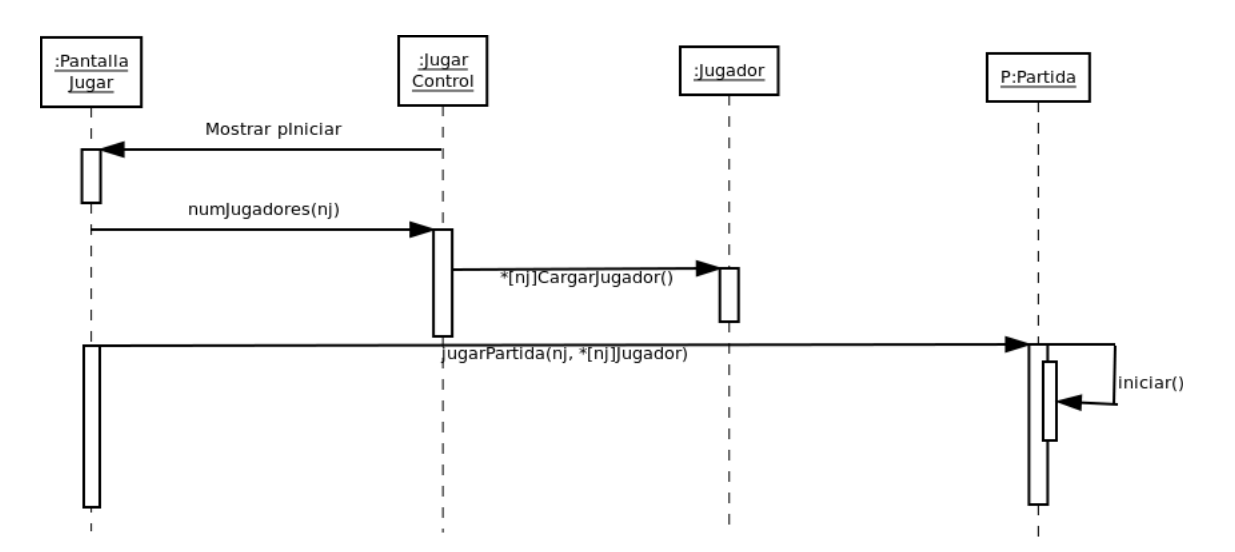


Figura 6.3: Diagrama de secuencia del caso de uso Jugar

6.5. Los menús, el gestor de partida y el gestor de jugadores

6.5.1. Diagramas de interacción

Diagrama de interacción del menú principal

Al entrar en la aplicación se muestra el menú principal:



Figura 6.4: Menú principal de FreeRummi

A continuación se expone el diagrama de interacción:

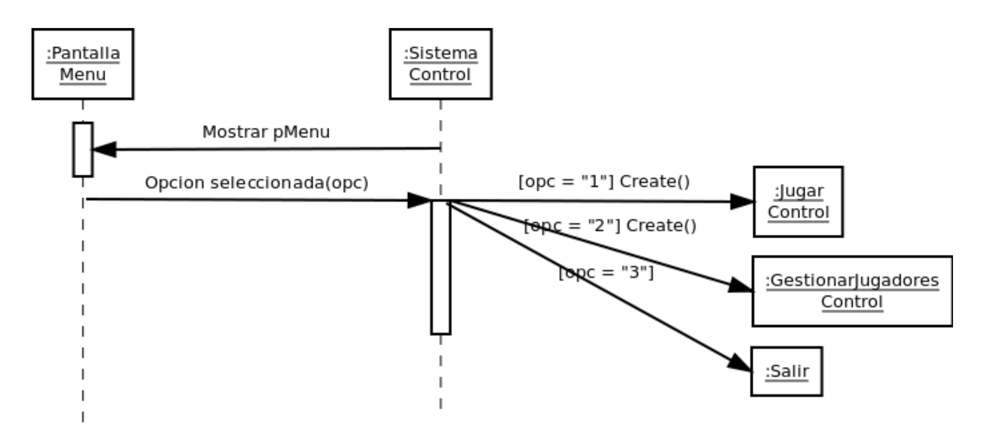


Figura 6.5: Diagrama de interacción del menú principal

- Para ejecutar la aplicación, se crea un objeto de la clase *Sistema Control (new.SistemaControl)* y se ejecuta una operación de la clase *Sistema Control* que invoca a la operación *Mostrar pMenu* de la clase *Pantalla Control*. La operación *Mostrar pMenu* se encarga de mostrar la pantalla del menú de la aplicación.

- Una vez mostrada la pantalla del menú de opciones se está a la espera de que se produzca algún evento. En este caso el evento que puede producirse es que el usuario seleccione una determinada opción del menú *(Opcion Seleccionada(opc))*

- En función de la opción del menú que el usuario seleccione, se crea un objeto de la clase de control que controla la acción asociada a la opcion (caso de uso) seleccionada.

Diagrama de interacción del submenú jugar

Este es el aspecto de la pantalla del submenú jugar:



Figura 6.6: Submenú Jugar de FreeRummi

Y el diagrama de interacción asociado:

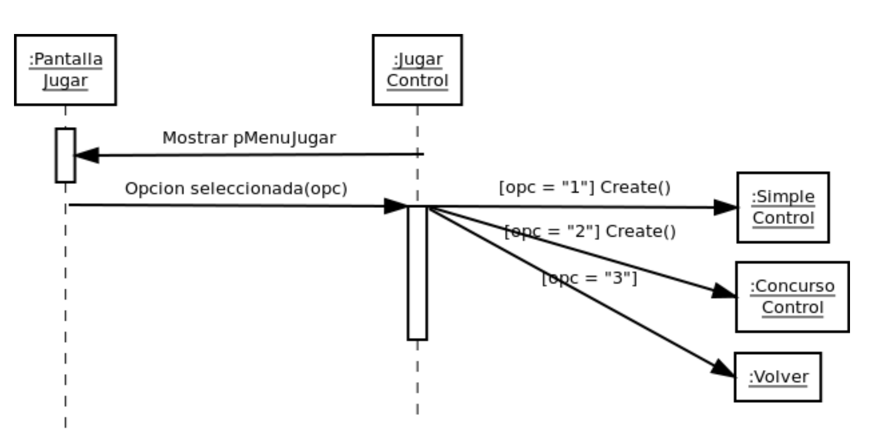


Figura 6.7: Diagrama de interacción del submenú Jugar

- Si en el menú principal se escogió la opción 1, se crea un objeto de la clase *Jugar Control (new.Jugar Control)* y se ejecuta una operación de la clase *Jugar Control* que invoca a la operación *Mostrar pMenuJugar* de la clase *PantallaJugar Control*. La operación *Mostrar pMenuJugar* se encarga de mostrar la pantalla del submenú jugar de la aplicación.

- Una vez mostrada la pantalla del menú de opciones se está a la espera de que se produzca algún evento. En este caso el evento que puede producirse es que el usuario seleccione una determinada opción del menú *(Opcion Seleccionada(opc))*

- En función de la opción del menú que el usuario seleccione, se crea un objeto de la clase de control que controla la acción asociada a la opcion (caso de uso) seleccionada.

Diagrama de interacción del submenú editar jugadores

Este es el aspecto de la pantalla del submenú editar jugadores:



Figura 6.8: Submenú Editar jugadores de FreeRummi

Y el diagrama de interacción asociado:

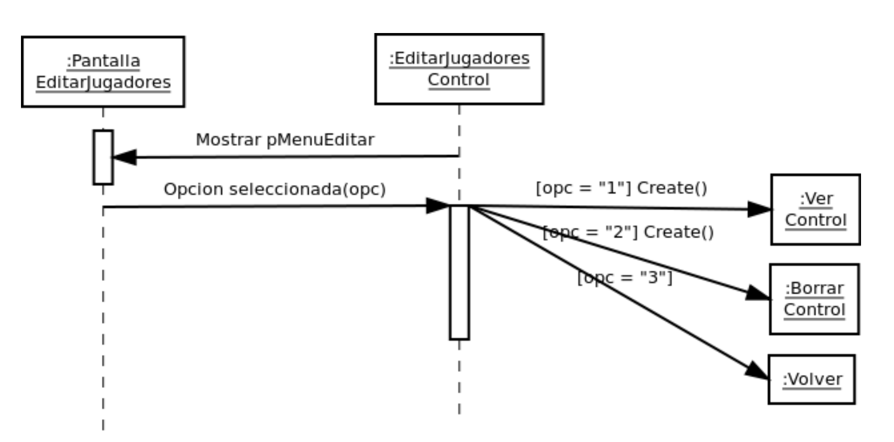


Figura 6.9: Diagrama de interacción del submenú Editar Jugadores

- Si en el menú principal se escogió la opción 2, se crea un objeto de la clase *EditarJugadores Control (new.EditarJugadores Control)* y se ejecuta una operación de la clase *EditarJugadores Control* que invoca a la operación *Mostrar pMenuEditar* de la clase *PantallaEditar Control*. La operación *Mostrar pMenuEditar* se encarga de mostrar la pantalla del submenú jugar de la aplicación.

- Una vez mostrada la pantalla del menú de opciones se está a la espera de que se produzca algún evento. En este caso el evento que puede producirse es que el usuario seleccione una determinada opción del menú *(Opcion Seleccionada(opc))*

- En función de la opción del menú que el usuario seleccione, se crea un objeto de la clase de control que controla la acción asociada a la opcion (caso de uso) seleccionada.

6.5.2. Diagramas de secuencia de los casos de uso

El gestor de jugadores presenta los siguientes diagramas de secuencia:

Diagrama de secuencia del caso de uso VerJugador

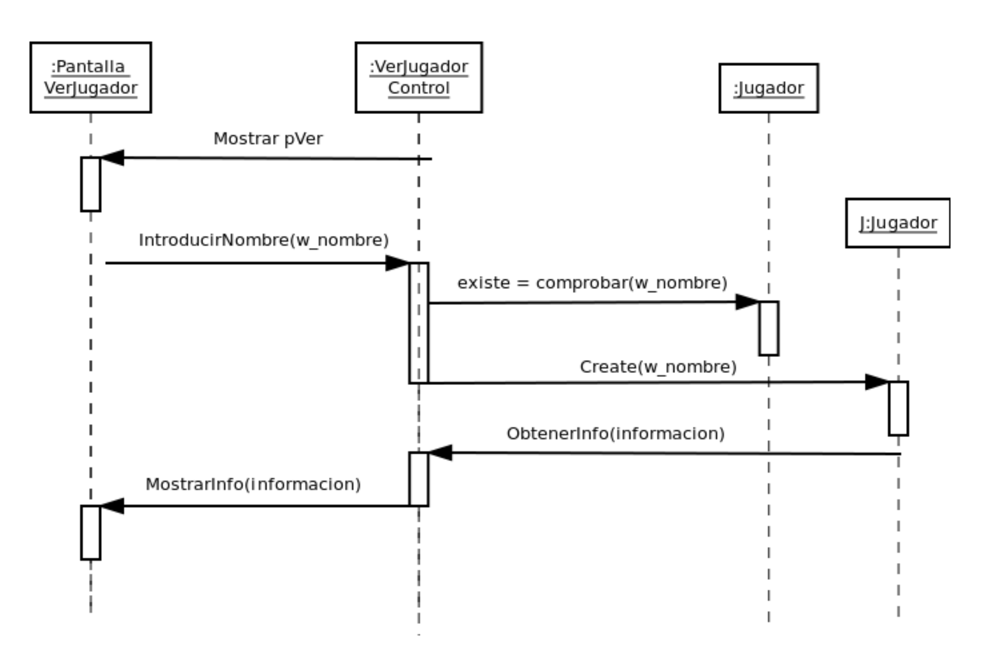


Figura 6.10: Diagrama de secuencia del caso de uso Ver Jugador

Diagrama de secuencia del caso de uso BorrarJugador

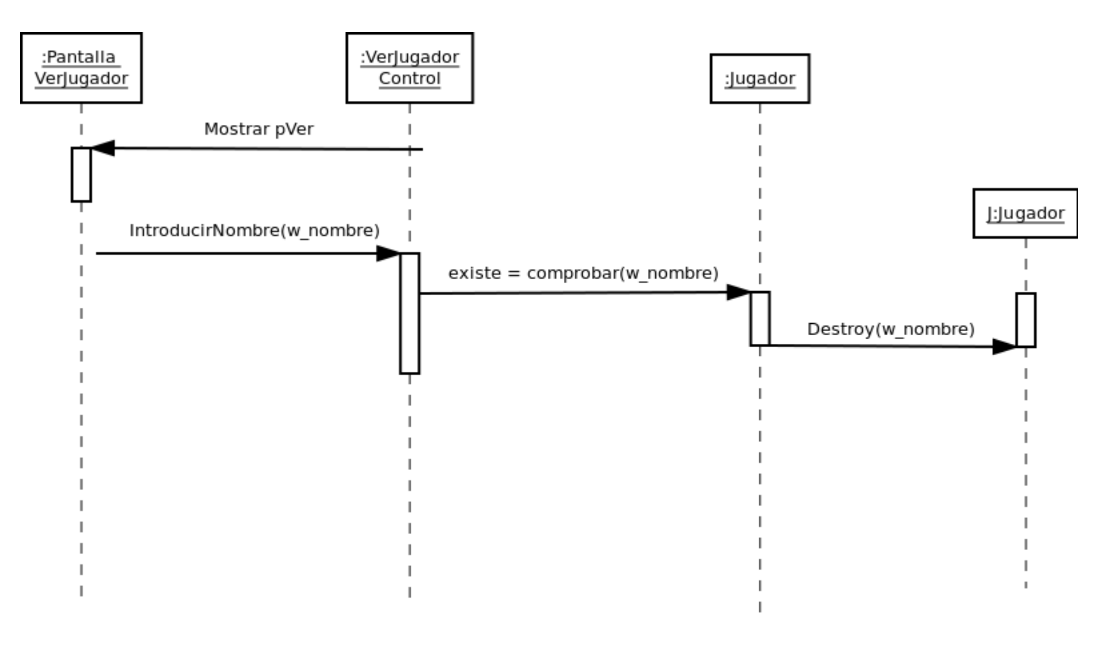


Figura 6.11: Diagrama de secuencia del caso de uso Borrar Jugador

6.6. Sistemas expertos

Esta sección es una de las más interesantes, ya que aquí es donde se verá como implementar los sistemas expertos que compondrán la inteligencia artificial de *FreeRummi*.

En primer lugar, son necesarias las funciones que permitan hacer combinaciones de escaleras y de conjuntos o que permitan insertar en el tablero fichas sueltas que sean válidas con las ya colocadas en el tablero.

Una vez tengamos esas funciones, Podremos realizar sistemas expertos como los que he creado de ejemplo. A continuación se muestran unos diagramas de flujo, en los que se puede observar como he diseñado los algoritmos de sistemas expertos.

- **Funcionamiento del primer sistema experto**: Si **está** abierto, intenta poner fichas sueltas, en caso de no conseguirlo intenta poner conjuntos y en caso de no conseguirlo intenta poner escaleras. Si **no está** abierto, intenta poner conjuntos, y si estos no suman más de 30 puntos intenta poner escaleras.

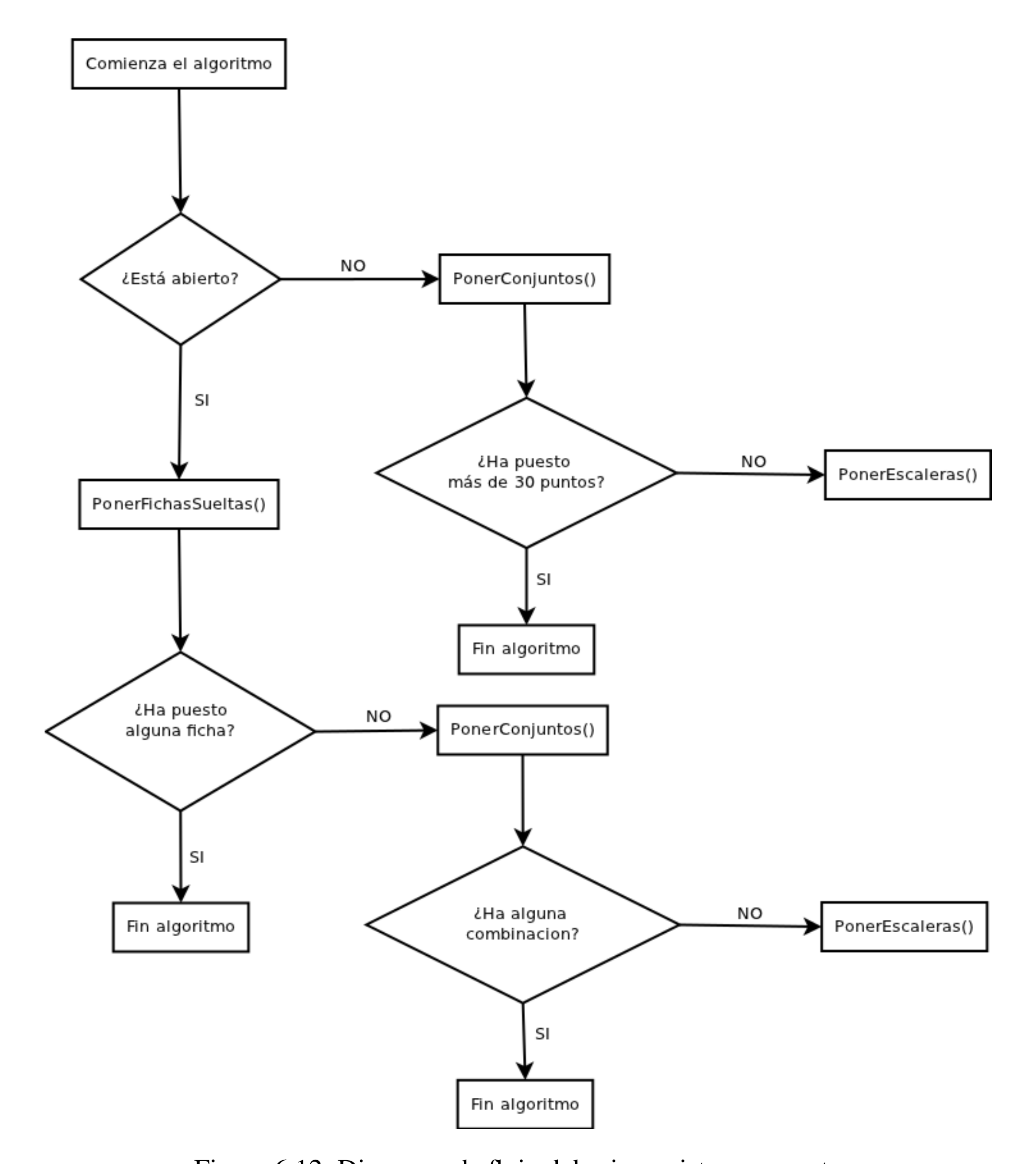


Figura 6.12: Diagrama de flujo del primer sistema experto

- **Funcionamiento del segundo sistema experto**: Si **está** abierto y tiene más de 40 puntos o menos de 20 intenta quitarse el mayor número de fichas poniendo escaleras y conjuntos, si tiene entre 40 y 20 puntos intenta poner fichas sueltas. Si **no está** abierto intenta poner tanto escaleras como conjuntos. Este robot intenta quedarse rápidamente con pocas fichas y cuando tiene muy pocos puntos (20), intenta ganar poniéndolo todo.

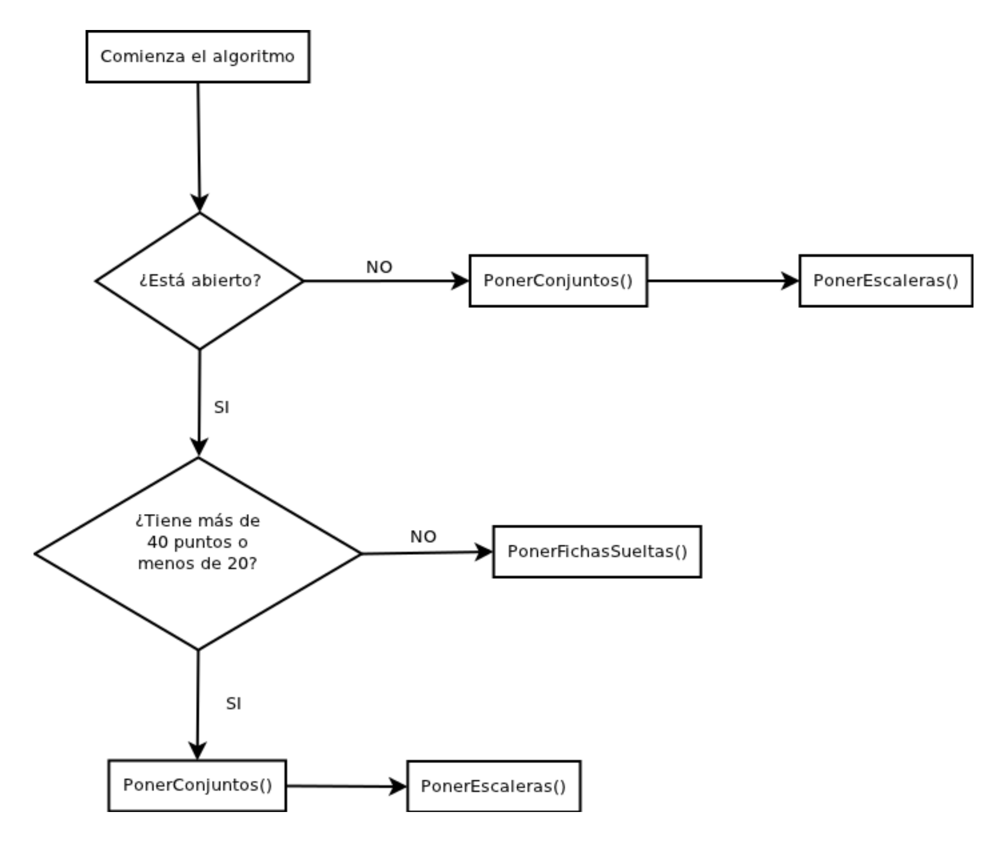


Figura 6.13: Diagrama de flujo del segundo sistema experto

- **Funcionamiento del tercer sistema experto**: Si **está** abierto y tiene más de 3 fichas, intenta poner conjuntos y escaleras y si tiene 3 o menos fichas intenta poner fichas sueltas. Si **no está** abierto intenta escaleras y si con ellas no suma más de 30 puntos intenta poner conjuntos. Este robot intenta quitarse rápidamente fichas y cuando solo tiene 3 juega con el tablero, de esta forma, no modifica los conjuntos del tablero hasta que no va a ganar.

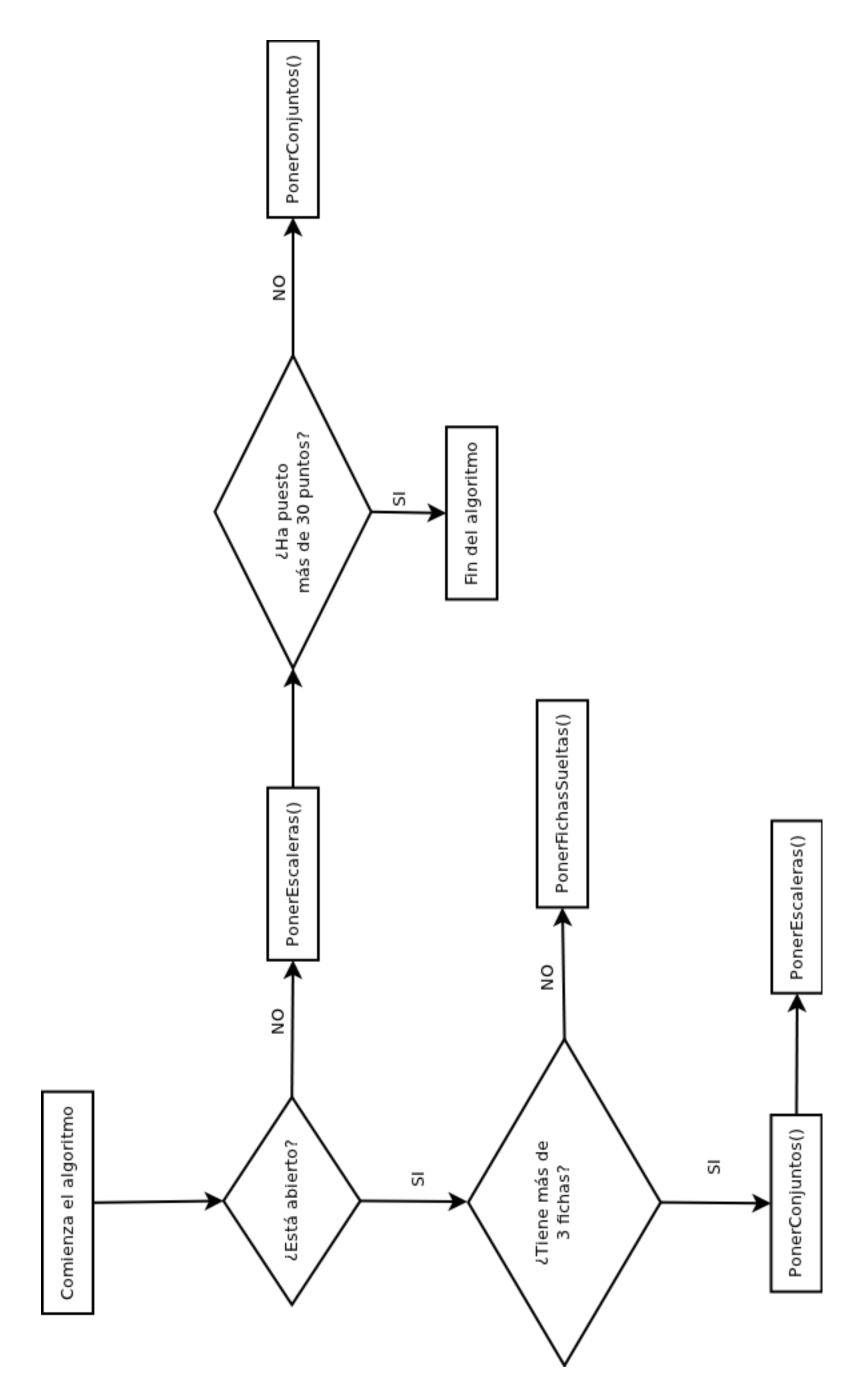


Figura 6.14: Diagrama de flujo del tercer sistema experto

Capítulo 7

Implementación

Una vez finalizados el análisis y el diseño, queda codificar y probar todo lo estudiado anteriormente. Para ello voy a utilizar como lenguaje de programación *C++* y como librería para entorno gráfico *libSDL*.

Según la siguiente extracción de wikipedia, *C++* es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades de programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

El motivo personal por el que he elegido este lenguaje, es por el conocimiento que poseo del mismo, debido a que muchas de las asignaturas que forman parte de mi carrera lo usan.

En cuanto a *libSDL* o Simple DirectMedia Layer (SDL) es un conjunto de bibliotecas desarrolladas con el lenguaje C que proporcionan funciones básicas para realizar operaciones de dibujado 2D, gestión de efectos de sonido y música, y carga y gestión de imágenes.
Además se han desarrollado una serie de bibliotecas adicionales que complementan las funcionalidades y capacidades de la biblioteca base:

- *SDL Mixer*: Extiende las capacidades de SDL para la gestión y uso de sonido y música en aplicaciones y juegos. Soporta formatos de sonido como Wave, MP3 y OGG, y formatos de música como MOD, S3M, IT, y XM.

- *SDL Image*: Extiende notablemente las capacidades para trabajar con diferentes formatos de imagen. Los formatos soportados son los siguientes: BMP, JPEG, TIFF, PNG, PNM, PCX, XPM, LBM, GIF, y TGA.

- *SDL TTF*: Permite usar fuentes TrueType en aplicaciones SDL.

He usado esta librería y no cualquier otra para el tema de los gráficos por que, además de que ya había trabajado con ella en la asignatura Diseño de Videojuegos, existe mucha documentación, tutoriales y ejemplos que son de gran ayuda ante posibles dudas que puedan aparecer.

7.1. La clase Cronometro

La función de esta clase es la de controlar el tiempo durante la partida.

Según las reglas del juego, el tiempo de un turno de *FreeRummi* es de un minuto. En principio esta clase solo iniciaba y reiniciaba el cronómetro tantas veces como turnos durara la partida, pero tras varias pruebas de funcionamiento, se propuso la posibilidad de pausar el juego, lo cual tras un breve estudio, se vio factible y se incluyó, teniendo así una clase más completa.

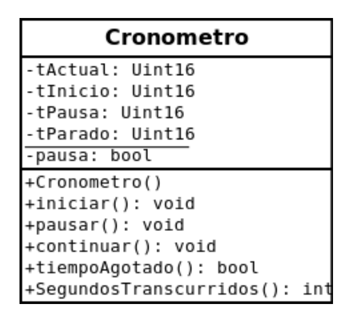


Figura 7.1: Clase Cronometro

Como se puede comprobar en el esquema de la figura 7.1, esta clase posee varios atributos para controlar los tiempos:

- `tInicio`: contiene el valor que devuelve la función `SDL_GetTicks()`, su función es almacenar el instante en el que se inició un determinado turno.
- `tActual`: contiene el tiempo actual, también se obtiene con `SDL_GetTicks()`, no es una variable que sirva por si sola, pero es necesaria para obtener otras.
- `tPausa`: contiene el instante en el que se entró en pausa, se obtiene con `SDL_GetTicks()`.
- `tParado`: indica el tiempo que ha estado pausado el cronometro, en principio su valor es 0 y en las sucesivas pausas se va obteniendo su valor de la forma:

  ```
  tParado += tActual - tPausa
  ```

Los métodos de esta clase son bastante simples, siendo la mayoría de ellos asignaciones a los atributos. Los únicos un poco más complejos son el método `bool tiempoAgotado()` y el método `int segundosTranscurridos()`:

- `bool tiempoAgotado()`: este método es el encargado de comprobar si el tiempo de un turno se ha agotado a no, la dificultad está en restar los tiempos de pausa. Por ello si el cronometro está en pausa, se devuelve el valor *0* simbolizando que nunca acabará el tiempo. Si no lo está devolverá verdadero o falso en función a la siguiente expresión:

  ```
  (tActual - tInicio) >60000 + tParado
  ```

  La cual comprueba que el tiempo transcurrido ha superado los 60 segundos, contando también con el tiempo que ha estado pausado el cronometro.

- `int segundosTranscurridos()`: este método convierte el formato que devuelve *SDL*, a segundos con la expresión:

  `(tActual- tParado) / 1000) - (tInicio / 1000)`

  Esta expresión, pasa a segundos el tiempo transcurrido y se lo resta al momento en el que se inició el cronometro, obteniendo así el tiempo que ha pasado.

7.2. La clase Ficha

Esta es una de las clases base del juego, la esencial en la partida. *FreeRummi*, está compuesto por un total de 106 fichas, 104 numeradas y 2 comodines. Las fichas numeradas lo están del 1 al 13 en cuatro colores: verde, azul, rojo y negro, teniendo 2 fichas del mismo número y color.

```
Ficha
-fComodin: bool
-num: unsigned
-col: Color
-imagen: Imagen *
+Ficha(comodin:bool=false,n:const unsigned=1,
       c:Color=VERDE)
+Ficha(f:const Ficha&)
+operator =(const Ficha&): Ficha&
+modificarComodin(n:const int,c:const Color): boo
+numero(): const unsigned
+color(): const Color
+comodin(): bool
+fichaNula(): bool
+posicionar(x:int,y:int): void
+mostrarFicha(v:Video *): void
+~Ficha()
-obtenerRuta(): const char*
```

Figura 7.2: Clase Ficha

Los atributos de esta clase son bastante simples : `num` para almacenar el número de la ficha y `col` para almacenar el color. Para los colores he definido la siguiente enumeración:

`enum Color{VERDE = 0, AZUL, ROJO, NEGRO}`

Con el atributo `fComodin`, sabremos si una determinada ficha es un comodín o no. Por último `imagen` es un puntero a un objeto de la clase `Imagen` necesario para la representación gráfica de la ficha.

Pasando ahora al tema de los métodos, además de los constructores y las funciones observadoras triviales, nos encontramos con los siguientes métodos:

- `modificarComodin(const int n, const Color c)`: Este método es utilizado para modificar los valores internos de una ficha que es comodín.
- `bool fichaNula() const`: Este método devuelve si una ficha es nula o no. Una ficha será nula cuando su valor numérico sea *0*. La función de la ficha nula es servir de control en el soporte o el tablero, para saber donde hay y donde no hay ficha insertada.

Además de estos métodos, la clase ficha posee dos métodos públicos para su representación gráfica y uno más privado para obtener la ruta donde se encuentra el archivo de imagen asociado a cada objeto de la clase Ficha:

- `void mostrarFicha(Video *v)`: Método para mostrar la ficha en la superficie que controla el objeto de la clase vídeo.

- `void posicionar(int x, int y)`: Método para modificar la posición en la que aparecerá la ficha en la superficie de vídeo.

Por último la clase ficha posee también varios operadores lógicos: igualdad, desigualdad y menor que, los cuales son utilizados por otras clases para tener un orden en las fichas.

7.3. La clase Fuente

Esta clase nos permite dibujar en una superficie SDL cualquiera una palabra o un texto que queramos mostrar a partir de una fuente TTF o de una rejilla de letras tipográficas.

```
Fuente
-fuente: TTF_Font *
-texto: SDL_Surface *
-color: SDL_Color
-posicion: SDL_Rect
+Fuente(ruta:const char*,tam:int)
+negrita(): void
+cursiva(): void
+subrayado(): void
+normal(): void
+mostrarTexto(v:Video *,frase:const char *
             x:int,y:int,r:int=0,g:int=0,
             b:int=0): void
+~Fuente()
```

Figura 7.3: Clase Fuente

Comenzaré repasando los atributos de esta clase:

- `fuente`: Este atributo es el encargado de almacenar la información de la fuente escogida, para ello se hace uso de la función SDL : `fuente = TTF_OpenFont(ruta, tamaño)`, donde la *ruta* contiene el directorio donde se encuentra la tipografía que se quiere cargar y el *tamaño* el tamaño en puntos que tendrá la imagen del texto.

- `texto`: Este atributo almacenará la imagen de la fuente a dibujar en la superficie.

- `color`: Este atributo servirá para poder cambiar el color a la fuente.

- `posicion`: Este atributo, como su propio nombre indica, servirá para posicionar la imagen del texto en la superficie de la pantalla.

Por otro lado, la clase ofrece varios métodos para personalizar, tales como `negrita()`, `cursiva()` y `subrayado()`, que permiten dar distintos aspectos al texto.
Y por último con el método `void mostrarTexto(Video *v, const char* frase, int x, int y, int r = 0, int g = 0, int b = 0)`, además de mostrar y posicionar la frase en la pantalla, podremos hacer que aparezca con el color que se haya definido en los tres últimos parametros *rgb*.

7.4. La clase GestorJugadores

Esta clase es la encargada de mostrar los menús para **Ver las estadísticas** de los jugadores así como para **Borrar jugadores**.

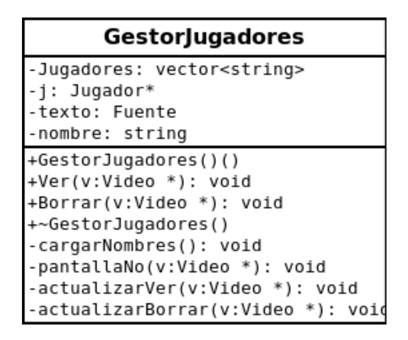


Figura 7.4: Clase Gestor de Jugadores

En el diagrama de la figura 7.4, se puede observar que la función de esta clase es la de hacer de interfaz para poder gestionar a los jugadores del sistema.

- En el constructor `GestorJugadores()`, se inicializan y posicionan las imágenes que se verán por pantalla, así como las fuentes que mostrarán los textos.
- La forma de actuar el método `void ver(Video *v)` es la siguiente: crea e inicia una banda sonora, carga en memoria el nombre de todos los jugadores del sistema y a continuación se pueden dar dos posibles ramas:
  - Que no haya jugadores en el sistema: el método entra en un bucle y muestra por pantalla un mensaje informando de que no hay jugadores para mostrar, ayudado del método privado `void pantallaNo(Video *v)` y a esperas de que se produzca un evento para volver al menú principal, en este caso hacer clic sobre el botón *cancelar*.
  - Que haya jugadores en el sistema: en este caso entra en un bucle en el que muestra la información del primer jugador con ayuda del método privado `actualizarVer(Video *v)` y a esperas de que se produzca un evento, que puede ser moverse por el menú para ver las estadísticas de otros jugadores o hacer clic sobre el botón *cancelar* para volver al menú principal.
- La forma de actuar el método `void Borrar(Video *v)` es similar a la de `Ver`: crea e inicia una banda sonora, carga en memoria el nombre de todos los jugadores del sistema y a continuación se pueden dar dos posibles ramas:
  - Que no haya jugadores en el sistema: el método entra en un bucle y muestra por pantalla un mensaje informando de que no hay jugadores para mostrar, ayudado del método privado `void pantallaNo(Video *v)` y a esperas de que se produzca un evento para volver al menú principal, en este caso hacer clic sobre el botón *cancelar*.
  - Que haya jugadores en el sistema: en este caso entra en un bucle en el que muestra el nombre del jugador seleccionado para borrar con ayuda del método privado `actualizarBorrar(Video *v)` y a esperas de que se produzca un evento, que puede ser moverse por el menú para seleccionar otros jugadores, hacer clic en el botón *confirmar* para borrar el jugador seleccionado o hacer clic sobre el botón *cancelar* para volver al menú principal.

7.5. La clase GestionPartida

La clase `GestorPartida`, es la encargada de mostrar la interfaz para introducir jugadores en la partida así como para manejar los distintos tipos de partida que se pueden jugar.

```
GestorPartida
-texto: Fuente
-ronda: Texto
-musica: Musica
-nombre: string
-numJugadores: unsgined
-partida: Partida *
-Jugadores: vector<string>
+GestorPartida()
+partidaSimple(v:Video *): void
+concurso(v:Video *): void
+~GestorPartida()
-seleccionJugadores(v:Video *): void
-seleccionJugadorSimple(v:Video *): void
-actualizarPantalla(v:Video *,jugActual:unsigned
                    zona:int,nom:string): void
-caracter(evento:SDL_Event): char
-reservado(nombre:string): bool
-desordenarJugadores(): void
-incrementarPosicion(): void
-pantallaRonda(r:const unsigned,v:Video *): void
```

Figura 7.5: Clase Gestor de partida

- En el constructor `GestorPartida()`, se inicializan y posicionan las imágenes que se verán por pantalla, así como las fuentes que mostrarán los textos.
- El método `void partidaSimple(Video *v)` tiene el siguiente esquema:
  1. Inicia la banda sonora.
  2. Selecciona los jugadores con el método `void seleccionJugadorSimple(Video * v)`.
  3. Crea la partida y añade los jugadores seleccionados.
  4. Se juega la partida.
  5. Se destruye la partida.
- El método `void concurso(Video *v)` tiene el siguiente esquema:
  1. Inicia la banda sonora.
  2. Selecciona los jugadores con el método `void seleccionJugadores(Video * v)`.
  3. Desordena el vector que contiene los jugadores.
  4. Crea la partida y añade los jugadores seleccionados.
  5. Se juegan tantas partidas como jugadores haya.
  6. Se destruye la partida.
- Los métodos para seleccionar los jugadores funcionan igual los dos, con la única diferencia de que en `void seleccionJugadorSimple()`, solo se escoge un único jugador, mientras que en `void seleccionJugadores(Video * v)` se escogen entre 2 y 4 jugadores, el esquema de ambos es el siguiente:
  1. Inicializa las variables auxiliares.

2. Se muestra un menú para seleccionar el número de jugadores que participarán con ayuda del método `void actualizarPantalla()`.
3. Se muestra un menú para introducir los jugadores:
   a) Si se va a jugar una partida simple, se muestra un menú para escribir el nombre del jugador que se desea añadir, se introduce y el resto de jugadores los carga el sistema como robots.
   b) Si se va a jugar un concurso, se entra en un bucle y se muestra un menú para ir introduciendo el nombre de los jugadores que participarán.

La diferencia entre elegir *partida simple* o *concurso* es, que en la *partida simple* las puntuaciones que se obtenga no contarán y se jugará contra el ordenador, es una partida de entrenamiento; mientras que en el *concurso*, las puntuaciones serán tenidas en cuenta, se jugarán varias partidas por concurso y se jugará contra jugadores humanos.

7.6. La clase IA

La clase IA es la encargada de manejar los sistemas expertos para poder jugar partidas contra el ordenador.

```
IA
-sCopia: Soporte
-tCopia: Tablero
-inteligencias: vector<int>
-Conjunto: vector<Ficha>
+IA()
+selectorIA(s:Soporte *,t:Tablero *,numInteligencias:int): i
+roboti(s:Soporte *,t:Tablero *): int
+robotii(s:Soporte *,t:Tablero *): int
+rabotiii(s:Soporte *,t:Tablero *): int
+~IA()
-ponerConjunto(): int
-ponerEscalera(): int
-ponerFichas(): int
-comprobarFicha(f:const Ficha&): bool
-colorRepetido(f:const Ficha&): bool
-insertarConjunto(): void
```

Figura 7.6: Clase IA

Los métodos públicos de esta clase están orientados a crear sistemas expertos a partir de los métodos privados:

- `int selectorIA()`: este método asigna un sistema experto a cada robot que controla el ordenador, de esta forma es más complicado saber que hará.

- `int roboti(Soporte *s, Tablero *t)`: sistema experto número uno: Si ESTÁ abierto, intenta poner fichas sueltas, en caso de no conseguirlo intenta poner conjuntos y en caso de no conseguirlo intenta poner escaleras. Si NO ESTÁ abierto, intenta poner conjuntos, y si estos no suman más de 30 puntos intenta poner escaleras.

- `int robotii(Soporte *s, Tablero *t)`: sistema experto número dos: Si ESTÁ abierto y tiene más de 40 puntos o menos de 20 intenta quitarse el mayor número de fichas poniendo escaleras y conjuntos, si tiene entre 40 y 20 puntos intenta poner fichas sueltas. Si NO ESTÁ abierto intenta poner tanto escaleras como conjuntos. Este robot intenta quedarse rápidamente con pocas fichas y cuando tiene muy pocos puntos (20), intenta ganar poniéndolo todo.

- `int robotiii(Soporte *s, Tablero *t)`: sistema experto número dos: Si ESTÁ abierto y tiene más de 3 fichas, intenta poner conjuntos y escaleras y si tiene 3 o menos fichas intenta poner fichas sueltas. Si NO ESTÁ abierto intenta escaleras y si con ellas no suma más de 30 puntos intenta poner conjuntos. Este robot intenta quitarse rápidamente fichas y cuando solo tiene 3 juega con el tablero, de esta forma, no modifica los conjuntos del tablero hasta que no va a ganar.

El código de estos métodos son simplemente llamadas a otros métodos de la clase que son los que realmente realizan las funciones de extracción y colocación de fichas:

- `int ponerConjunto()`: como su propio nombre indica, la función de este método es colocar en el tablero todos los conjuntos válidos que posea el soporte, haciendo uso de los métodos auxiliares `bool comprobarFicha()` y `bool colorRepetido()`
- `int ponerEscalera()`: la función de este método es colocar en el tablero todas las escaleras válidas que posea el soporte, haciendo uso de los métodos auxiliares `bool comprobarFicha()` y `bool colorRepetido()`
- `int ponerFichas()`: la función de este método es colocar en el tablero todas las fichas sueltas que se puedan colocar, haciendo uso de los métodos auxiliares `bool comprobarFicha()` y `bool colorRepetido()`

7.7. La clase Imagen

Esta clase se encarga de cargar una imagen que esté en cualquier formato compatible con *SDL_Image*.

Imagen
-imagen: SDL_Surface * -posicion: SDL_Rect
+Imagen(ruta:const char *,x:Sint16=0,y:Sint16=0 +dibujar(v:Video *): void +modificarX(x_:Sint16): void +modificarY(y_:Sint16): void +obtenerX(): const Sint16 +obtenerY(): const Sint16 +ancho(): const Uint16 +alto(): const Uint16 +~Imagen()

Figura 7.7: Clase Imagen

Como se observa en la figura 7.7, esta clase tiene unos métodos bastante triviales, para modificar y obtener información acerca de la imagen.

7.8. La clase Iniciado

Esta clase simplemente inicializa la librería *SDL*.

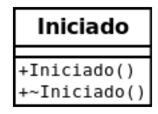


Figura 7.8: Clase Iniciado

El *constructor* de esta clase sigue el siguiente esquema:

1. Inicializa el sistema de video.
2. Inicializa el sistema de audio *Mixer*.
3. Inicializa el sistema para el manejo de fuentes *TTF*.

Mientras que el *destructor* destruye con el siguiente esquema:

1. Destruye el sistema de fuentes.
2. Destruye el sistema de audio *Mixer*.
3. Destruye el sistema video.

7.9. La clase Jugador

Esta clase es la encargada de gestionar a los jugadores así como guardar sus datos en disco y cargarlos en distintas sesiones.

```
Jugador
+Rango: enum = {NOVEL = 1, MEDIO, AVANZADO, EXPERTO
-robot: bool
-nom: string
-puntPartida
-puntGeneral: int
-partGanadas: int
-partPerdidas: int
-partAbandonadas
-porcentajeGanadas: double
-porcentajePerdidas: double
-porcentajeAbandonadas: double
-rango: Rango
-soporte: Soporte
+Jugador(nombre:const string&,robt:bool=false)
+guardarJugador(): void
+borrarJugador(): void
+partidaGanada(): void
+partidaPerdida(): void
+partidaAbandonada(): void
+puntuacionPartida(puntos:int,situacion:int): void
+iniciarSoporte(m:Monton&): void
+obtenerSoporte(): Soporte&
+nombre(): const string
+puntuacionPartida(): const int
+partidasGanadas(): const int
+partidasPerdidas(): const int
+partidasAbandonadas(): const int
+porGanadas(): const double
+porPerdidas(): const double
+porAbandonadas(): const double
+rangoJugador(): const Rango
+jRobot(): bool
```

Figura 7.9: Clase Jugador

Repasaré los métodos más interesantes de esta clase, que son los relacionados con el almacenamiento en memoria secundaria:

- El constructor `Jugador(const string& nombre, bool robt = false)`, sigue el siguiente esquema:

    1. Si el jugador es un robot simplemente inicializa las variables con valor nulo ya que de los robots no se guardará la información.

2. En caso contrario, tras cargar en memoria los nombres de los jugadores del archivo */free-rummi/guardado/gestorJugadores.sav*, se pueden dar dos casos, que el nombre introducido como primer parámetro exista en el sistema y que no exista:

   a) Si no existe, se crea un fichero en memoria secundaria, con el formato *nombre.sav* y se escriben en el, su nombre y su puntuación general, partidas ganadas, perdidas y abandonadas, que logicamente serán *0*. A continuación se inicializan todos sus atributos.

   b) Si existe, se abre el fichero *nombre.sav* y se inicializan sus atributos con los datos que hay en dicho fichero.

Como se puede observar, el constructor sirve tanto para *crear* como para *cargar* jugadores.

- El método `void guardarJugador()` sigue el siguiente esquema:

  1. Comprueba que el jugador NO es un robot.
  2. Busca que el jugador efectivamente tiene un fichero en memoria secundaria.
  3. Actualiza los datos que existían en el fichero con los nuevos.

- El método `void borrarJugador()`, sigue el siguiente esquema:

  1. Busca que el jugador efectivamente tiene un fichero en memoria secundaria.
  2. Elimina de dicha memoria el fichero *nombre.sav* y a continuación borra la linea *nombre* del fichero *gestorJugadores.sav*

- El resto de métodos simplemente sirven para actualizar la información del jugador, tales como partidas ganadas, perdidas, puntuación general, etc.

7.10. La clase Menu

Esta clase es la encargada de mostrar los menús navegar por el juego: menú principal, menú jugar y menú jugadores.

```
Menu
-opcion: Efecto
-descripción: Fuente
+Menu()
+menuPpal(v:Video *): int
+menuJugar(v:Video *): int
+menuJugadores(v:Video *): int
+~Menu()
-actualizarPpal(v:Video *): void
-actualizaJugar(v:Video *): void
-actualizarJugadores(v:Video*): voi
```

Figura 7.10: Clase Menu

En el diagrama de la figura 7.10, se puede observar que la función de esta clase es la de hacer de interfaz para navegar por los menús del sistema

- En el constructor `Menu()`, se inicializan y posicionan las imágenes que se verán por pantalla, así como las fuentes que mostrarán los textos.

- La forma de actuar el método `int menuPpal(Video *v)` es la siguiente: crea e inicia una banda sonora, y a continuación se queda a la espera de que se produzca algún evento. Estos eventos pueden ser: seleccionar la opción **Jugar**, seleccionar la opción **Jugadores** o seleccionar la opción **Salir**. cada opción llevará a un submenú controlado por los métodos:

  - `int menuJugar(Video *v)`, el cual actúa de igual forma que su antecesor solo que con las opciones **Simple**, **Concurso** y **Volver**. Las dos primeras opciones desembocan en el Gestor de partida, mientras que la última vuelve al menú principal.
  - `int menuJugadores(Video *v)`, el cual actúa de igual forma que su antecesor solo que con las opciones **Ver**, **Jugar** y **Volver**. Las dos primeras opciones desembocan en el Gestor de jugadores, mientras que la última vuelve al menú principal.

- El resto de métodos privados, tienen la función de mostrar por pantalla las opciones y las descripciones de cada opción.

7.11. La clase Monton

Esta clase es una de las clases base de *FreeRummi*, la cual contiene el total de las fichas que componen el juego.

```
Monton
-fichRest: int
-sigExt: Ficha
-monton: vector<Ficha>
+Monton()
+extraer(): const Ficha&
+vacio(): bool
+~Monton()
-desordenar(): void
```

Figura 7.11: Clase Monton

Como se observa en la figura superior 7.11, esta clase posee un vector de *Fichas*. Sus métodos son bastante simples, los repasaré brevemente:

- El constructor, crea e inserta todas las fichas en el vector y a continuación ayudado del método privado `desordenar()`, obtiene una permutación aleatoria del mismo.
- El método `const Ficha& extraer()`, se encarga de sacar la ficha que se encuentra en la primera posición del vector.
- El método `bool vacio()`, simplemente informa de si el montón está vacío o no.

7.12. La clase Partida

Esta clase es el pilar central de la aplicación, es la encargada de gestionar la partida, con todos sus eventos.

```
Partida
-tablero: Tablero
-tableroCopia: Tablero
-crono: Cronometro
-monton: Monton
-Jugadores: vector<Jugador *>
-soporteCopia: Soporte
-numJugadores: unsigned
-jugadorActual: unsigned
-pPulsar: Puntero
-pSoltar: Puntero
-pMovimiento
-evento: SDL_Event
-f: Ficha
-robar: bool
-comprobar: bool
-salir: bool
-pausa: bool
-turn: int
-cont: int
-cron: Fuente
-info: Fuente
-cambio: Fuente
-musica: Musica
+Partida(numJug:unsigned=2)
+jugar(v:Video *,tipoPartida:int=0): voi
+anadirJugador(j:Jugador *): bool
+eliminarJugador(j:Jugador *): bool
+iniciarJugadores(): void
+~Partida()
-turno(v:Video *): int
-mostrarTextos(v:Video *): void
-actualizarPantalla(v:Video *): void
-pantallaCambio(v:Video *): void
-pantallaGanador(v:Video *): void
-pantallaAbandonado(v:Video *): void
-salirPartida(v:Video *): bool
-pantallaSalir(v:Video *): void
-eventosPulsacion(): bool
-eventosLiberacion(): bool
-eventosTeclado(): bool
```

Figura 7.12: Clase Partida

Explicaré que hacen los métodos mas relevantes de esta clase:

- El constructor de la clase, `Partida(unsigned numJugadores = 2)` inicializa todas la variables de imagen, de sonido y de textos. El parámetro que recibe contiene el número de jugadores que participarán en la partida, por defecto valdrá *2*.
- El primer método interesante es `void jugar(Video *v, int tipoPartida = 0)`. El segundo parámetro que recibe esta función tomará el valor 0 si es una partida simple o -1 en caso de ser un concurso. Este método contiene el bucle del juego el cual sigue el siguiente esquema:
  1. inicializa la banda sonora y los soportes de los jugadores, así como las variables auxiliares
  2. Entra en el bucle, el cual tiene como condición para salir que algún jugador haya ganado o abandonado la partida.
  3. Si el jugador es un robot, se hace uso de la clase IA.
  4. En caso contrario se pasa el control al método `turno()`.
  5. Dependiendo de lo que devuelva dicho método, se continua dentro del bucle o se sale. En caso de seguir dentro se incrementa el jugador actual y se vuelve al paso *2*.
  6. Una vez fuera del bucle, se muestra una pantalla de ganador o de abandonado, según el caso.
  7. Por último, si el tipo de partida es un concurso, se actualizan los datos de los jugadores con las puntuaciones obtenidas.

- El método `int turno()`, es el método que controla todos los eventos que se producen en la partida, tanto de teclado como de ratón, con ayuda de los métodos `bool eventosPulsación`, `bool eventosLiberacion` y `bool eventosTeclado`. Este método sigue el siguiente esquema:

  1. En primer lugar inicializa las variables auxiliares.
  2. A continuación entra en un bucle de captura de eventos en el que la condición para salir es que se agote el tiempo, se decida salir, se pulse el botón robar o el botón comprobar.
  3. A partir de este punto se pueden dar tres casos:
     a) *Se acabó el tiempo*: se comprueba que el tablero está bien y el jugador roba una ficha. Si el tablero no esa bien el jugador roba tres fichas.
     b) *El jugador pulsó robar*: se comprueba que el tablero está bien y el jugador roba una ficha. Si el tablero no esa bien el jugador roba tres fichas.
     c) *El jugador pulsó comprobar*: se comprueba que el tablero está bien. Si el tablero no esa bien el jugador roba tres fichas.
  4. Si el jugador pulsó salir, se devuelve -1, lo cual indica que el jugador abandona la partida.
  5. En caso contrario, se comprueba si el jugador tiene el soporte vacío, devuelve 1 o si aun no lo ha conseguido vaciar, devuelve 0.

  Para poder volver al estado anterior y no perder nunca el tablero válido y los soportes de los jugadores, he hecho uso de tableros y soportes copia, con los cuales si el jugador realizaba una acción indebida, siempre se puede volver al estado anterior.

  En cuanto al manejo de los eventos del ratón, he utilizado tres objetos de la clase Puntero, con lo cual siempre tenia la información de donde se pulsó, donde se soltó y que movimiento se produjo, por si el jugador realizaba algo indebido, poder dejar las fichas en los sitios de los que provenían.

El resto de métodos de esta clase son auxiliares a estos dos y sirven para añadir y eliminar jugadores de la partida, mostrar los elementos de la partida por pantalla y manejar los eventos que se produzcan.

7.13. La clase Puntero

Esta clase tiene el objetivo de controlar la posición del puntero del ratón en todo momento. Además permitirá dar respuesta a las acciones producidas en el ratón sobre las diferentes partes de la pantalla.

```
Puntero
-x: int
-y: int
-boton: int
+Puntero()
+operator =(p:const Puntero&): Puntero&
+actualizar(): void
+posX(): int
+posY(): int
+boton(): int
+~Puntero()
```

Figura 7.13: Clase Puntero

como se observa en el diagrama 7.13, los métodos y atributos de esta clase son muy simples por lo que no procede ninguna explicación.

7.14. La clase Efecto

Esta clase nos proporciona todo aquello que necesitamos para introducir sonidos de acción en la aplicación.

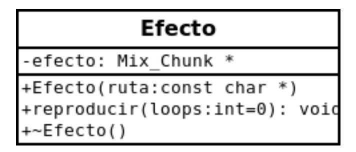


Figura 7.14: Clase Efecto

El funcionamiento es bastante simple, mediante el constructor se carga el sonido en memoria y para reproducir dicho sonido está la función `void reproducir(int loops = 0)`, cuyo parámetro indica el número de veces que se reproducirá el sonido, siendo 0 una vez y -1 infinito.

7.15. La clase Música

Esta clase nos permite dotar de este elemento al juego.
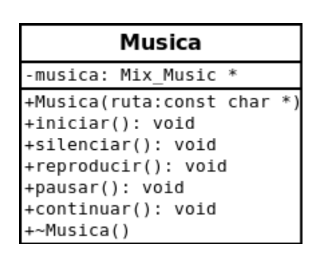


Figura 7.15: Clase Musica

Esta clase a sufrido varios cambios a medida que se iban implementando los incrementos: en un principio la clase solo tenia el método `reproducir()`, pero una vez que se introdujo la pausa a la clase `cronometro`, hubo que crear los métodos `pausar()` y `continuar()`, para poder crear el efecto de pausado completo. En un último incremento se añadió la variable global `bool silencio`, con la cual se puede dejar en silencio el juego desde cualquier menú, y para poder modificar esta variable, se creó el método `silenciar()`.

7.16. La clase Soporte

Esta es otra de las clases base de *FreeRummi*. Es la encargada de almacenar las fichas que un jugador posee.

```
Soporte
-fichRest: int
-suma: int
-sigExt: Ficha
-sop[28]: Ficha
-abrto: bool
+Soporte()
+Soporte(s:const Soporte&)
+operator =(s:const Soporte&): Soporte&
+escogerFicha(int pos): cons Ficha&
+ordenar(f:const Ficha&,pos:int): bool
+robar(m:Monton&): void
+abrir(): void
+ordenarNumero()(): void
+ordenarColor(): void
+vacio(): bool
+fichas(): const int
+sumaSoporte(): const int
+abierto(): bool
+~Soporte()
-void juntarFichas(): void
-intercambiar(pos1:const int,pos2:const int): voi
```

Figura 7.16: Clase Soporte

Esta clase también ha ido sufriendo cambios a lo largo de los incrementos, en principio decidí que la estructura de datos adecuada era un multiset para guardar las fichas ordenadas, pero eso no me daba opciones a ordenar el soporte como quisiera. Por eso cambié por un vector. Una vez tenia el vector podía intercambiar las fichas de posición, pero siempre intercambiando las posiciones de las fichas, sin posibilidad de poner las fichas en posiciones vacías. Realicé ese cambio y así se mantuvo un tiempo, hasta que empecé a implementar la clase `IA`, en la cual necesitaba ordenar el soporte por colores o por números, así que una vez que hice los métodos `void ordenarNumero()` y `void ordenarColor()`, pensé que podrían ser de utilidad para el usuario también.

7.17. La clase Tablero

Es la última clase básica de *FreeRummi* y se encarga de manejar el tablero de la partida y comprobar que todos los grupos que hay en el son correctos.

```
Tablero
-sigExt: Ficha
-tab[11][26]: Ficha
-conjunto: vector<Ficha>
+Tablero()
+Tablero(t:const Tablero&)
+operator =(t:const Tablero&)
+escogerFicha(posX:int,posY:int): const Ficha
+insertarFicha()(f:const Ficha&,posX:int,
               posY:int): bool
+mostrarTablero(v:Video *): void
+comprobar(): bool
+~Tablero()
-comprobarFicha(f:const Ficha&): bool
-colorRepetido(f:const Ficha&): bool
```

Figura 7.17: Clase Tablero

En esta clase nos encontramos el método `bool comprobar()`, el cual ayudado de los métodos auxiliares `bool comprobarFicha(const Ficha&)` y `bool colorRepetido(const Ficha&)`, comprueban si los grupos del tablero son correctos o no. El modo de funcionar de este método es el siguiente: comienza a comprobar por la primera casilla del tablero si hay ficha o no, cuando encuentra una

casilla con ficha, la comprueba con la anterior y si es válida sigue comprobando, si no lo es, devuelve false. Dependiendo del tipo de grupo que sea hará uso de uno o de los dos métodos auxiliares: para las escaleras solo necesita `comprobarFicha()`, pero para los conjuntos necesita tanto uno como otro.

7.18. La clase Video

Esta clase es la encargada de gestionar la superficie sobre la que veremos la aplicación.

```
Video
-ancho: int
-alto: int
-profundidad: int
-pantalla: SDL_Surface *
+Video(ancho:int=1024,alto:int=768,prof:int=32
       full:bool=false)
+superficie(): SDL_Surface *
+actualizar(): void
+anchoSup(): int
+altoSup(): int
+profSup(): int
+~Video()
```

Figura 7.18: Clase Video

Como se observa en la figura 7.18, los métodos de esta clase son todos triviales, explicaré por tanto el constructor de la clase. La aplicación está diseñada para ejecutarse en modo ventana con una resolución de 1024 x 768 x 32 o en formato pantalla completa, de lo cual se encarga el último parámetro del constructor `bool full = false`, que por defecto es falso. El resto de parametros indican las proporciones de la pantalla.

7.19. Otros detalles de implementación

Antes de finalizar el capítulo, voy a comentar la función del fichero de cabecera `constantes.h`. En el se encuentra una lista con todas las constantes de posición de todos y cada uno de los elementos gráficos del juego, en el siguiente orden:

- Posición en el eje x: ELEMENTO_X
- Posición en el eje y: ELEMENTO_Y
- Posición en el eje x más el ancho de la imagen: ELEMENTO_XW
- Posición en el eje y más el alto de la imagen: ELEMENTO_YH

La finalidad de este fichero, es poder cambiar cómodamente los valores en caso de querer realizar modificaciones en los gráficos, sin tener que buscar por el código.
Por otro lado, en las clases que tiene elemento gráfico, he añadido funciones cuyo prototipo es:

```
bool areaElemento(const Puntero& p)
```

cuyo propósito es el de confirmar si un evento determinado por el puntero p, se ha producido dentro del área del elemento o no.
Como cierre de este capítulo, quiero comentar que durante la elaboración del mismo no he querido entrar en detalles muy concretos de implementación, ya que adjunto la documentación de todo el código, la cual se puede encontrar en el directorio */freeRummi/doc* del CD y que durante la implementación fui añadiendo comentarios a todos los fragmentos de código que podían resultar costosos de entender.

Capítulo 8

Pruebas

8.1. Filosofía de las pruebas

El objetivo de las pruebas es la detección de defectos en el software y descubrir un defecto debería considerarse el éxito de una prueba.

Las recomendaciones de G. J. Myers [8] para la realización de las pruebas son las siguientes:

1. Cada caso de prueba debe especificar el resultado de salida esperado. Este resultado se compara con el obtenido en la prueba y las diferencias entre ambos se consideran síntomas de un posible defecto en el software.
2. El programador debe evitar probar sus propios programas.
3. Se debe analizar con detalle el resultado de cada prueba para poder detectar posibles defectos.
4. Al definir los casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como inválidos e inesperados.
5. Las pruebas deben centrarse en dos objetivos: probar si el software no hace lo que debe y probar si el software hace lo que no debe.
6. Los casos de prueba deben diseñarse y documentarse detalladamente.
7. No deben hacerse planes de prueba suponiendo que no hay defectos en los programas. Hay que asumir que siempre hay defectos y hay que detectarlos.
8. Donde hay un defecto hay otros, es muy probable que se descubran nuevos defectos donde ya se ha descubierto alguno.
9. Las pruebas son una tarea tanto o más creativa que el desarrollo de software.

8.2. Incremento 2: Organización de las partes básicas del juego

- *¿Se crean correctamente los objetos de las clases básicas (Ficha, Monton, Soporte y Tablero)?*

  Sí se crean. Se fueron probando una a una las clases con los distintos parámetros que iban necesitando. La clase montón en un principio no creaba bien todas las fichas y era debido a la función de desordenación que no trabajaba correctamente, por lo que una vez detectado el fallo se corrigió.

- *¿Funcionan correctamente los métodos modificadores de las clases?*

  Sí funcionan. Se probaron tanto en casos aislados: con datos de entrada válidos e inválidos; como en partidas completas.

- *¿Devuelven los observadores los datos correctamente?*

  Sí los devuelven. Como en el caso anterior, también se probaron en casos aislados y en partidas completas.

- *¿Se inicializan correctamente los soportes?*

  Sí se inicializan. Cada jugador tiene catorce fichas al iniciar la partida. Se comprueba que no hay fichas repetidas ni más fichas de la cuenta.

- *¿Se comprueba correctamente el tablero?*

  Sí. Es uno de los casos de prueba más estudiados, ya que de él depende el éxito del juego. En principio se fueron probando casos aislados tanto buenos como malos y se observaba que siempre devolvía lo correcto. Más tarde se probó en una partida completa con casos buenos y malos, con varios conjuntos y siempre devolvía lo correcto.

- *¿Se pierden fichas o algún otro elemento durante la ejecución de una partida?*

  No se pierden. Es uno de los defectos que más a menudo aparecían cuando se incluían nuevas funcionalidades: al robar la ficha no iba al soporte deseado, al insertar en el tablero la ficha desaparecía, al mover fichas dentro del soporte también desaparecían, etc. Pero se detectaron los problemas y se puso solución. Una vez solucionado, se han hecho todo tipo de pruebas intentando perder fichas y nunca se ha conseguido.

- *¿Se puede jugar una partida completa en la consola?*

  Sí se puede, pero con restricciones. Como ya comenté en el capítulo tres, en este incremento trataba de crear la base, ignorando muchas funcionalidades, cabe recordar también que en este incremento el juego se ejecutaba en consola. Por tanto, se puede jugar una partida completa pero sin muchos de los elementos que luego se fueron incluyendo (entorno gráfico, cronometro, etc).

- *¿Finaliza correctamente la ejecución del programa?*

  Sí. Tanto si se deja una partida a medias, como si se finaliza vaciando un soporte, el programa termina correctamente.

8.3. Incremento 3: Introducción de gráficos y sonidos

- *¿Se inicia correctamente SDL?*

  Sí se inicia. Se ve una ventana con las dimensiones dadas, en caso contrario daría un error.

- *¿Se inicia correctamente el subsistema de audio?*

  Sí. El subsistema de sonido se cargar correctamente ya que se puede escuchar la banda sonora de la partida.

- *¿Se inicia correctamente la librería TTF?*

  Sí. En la pantalla aparecen los textos de prueba que se hicieron para este incremento.

- *¿Se muestran correctamente las imágenes en la superficie?*

  Sí se muestran. Además es posible cargar imágenes en formato *PNG* y otros formatos de imagen.

- *¿Se posicionan las imágenes según las coordenadas que se le pasan?*

  Sí se posicionan. En principio incluía las coordenadas dentro del código, pero tenía el inconveniente de que cada vez que cambiaba algo de sitio tenia que buscarlo por el código, así que incluí el fichero `constantes.h` con todas las coordenadas de todos los elementos gráficos.

- *¿Se puede jugar en pantalla completa?*

  Sí se puede. El juego presenta respuesta a los eventos tanto en pantalla completa como en modo ventana.

- *¿Capta el sistema correctamente los eventos producidos por los periféricos de entrada?*

  Sí los capta, tanto los de teclado como los de ratón. Para ello se realizaron pequeños programas de prueba con cada uno de los periféricos y luego se insertaron en el programa, creando así la clase Puntero. En este incremento, las fichas no se pueden arrastrar, la forma de moverlas es pinchando en la ficha que se desee mover y luego pinchando en el destino que se quiere para la ficha, si es correcto el destino la ficha se mueve, si no, se queda donde estaba.

- *¿Desaparecen elementos gráficos durante la ejecución de una partida?*

  No desaparecen. En principio, casualmente desaparecían algunas imágenes de fichas y luego volvían a aparecer, pero se encontró el error y se corrigió.

- *¿El sonido funciona correctamente?*

  Sí funciona. Tanto los efectos como la banda sonora funcionan correctamente. Los efectos en principio se solapaban unos a los otros, no dejando sonar dos sonidos a la vez, pero se solucionó y ahora por ejemplo se tiene sonando el cronometro a la vez que el sonido de las fichas cuando se mueven o el que se produce al robar.

- *¿Finaliza correctamente la ejecución del programa?*

  Sí. Tras añadir las clases encargadas de manejar todo el entorno multimedia el juego sigue funcionando y finalizando correctamente.

8.4. Incremento 4: Diseño e implementación de la clase Partida

- *¿Funciona correctamente el cronometro?*

  Sí funciona. En principio cuando no había pausa el cronometro funcionaba perfectamente. Una vez se incluyó la pausa, cuando llegaba al final del turno el cronometro seguía contando tanto tiempo como hubiese estado en pausa, lo cual se detectó y se corrigió. Otro problema que presentaba era que estando en pausa, si se pasaban los sesenta segundos, se pasaba de turno, también se detectó y corrigió. Cuando se añadió la funcionalidad de la pantalla de confirmación de salida de la partida, el cronometro no presentó problema alguno.

- *¿Funciona correctamente el constructor de la clase Jugador?*

  Sí funciona. Se probaron distintas entradas, válidas e inválidas para el constructor y se probó que: si el jugador no existe, el sistema crea una entrada con su nombre en el fichero *gestorJugadores.sav* y crea un fichero *nombre.sav*, en el cual en la primera linea aparece el nombre del jugador, en la segunda su puntuación general, en la tercera las partidas ganadas, en la cuarta las partidas perdidas y en la quinta las partidas abandonadas. Si el jugador existe en el sistema lo busca en el fichero *gestorJugadores.sav* y carga en memoria los datos referentes al fichero *nombre.sav*. Tanto en un caso como en otro el sistema termina de inicializar los datos del jugador en memoria.

- *¿Se guardan correctamente los datos de los jugadores?*

  Sí se guardan. El sistema abre el fichero correspondiente al jugador y escribe en el los datos actualizados.

- *¿Se borran correctamente los jugadores del sistema?*

  Sí se borran. El sistema elimina la entrada del jugador del archivo *gestoJugadores.sav* y elimina el archivo *nombre.sav*.

- *¿Se actualizan correctamente las puntuaciones de los jugadores?*

  Sí. Además cada vez que se actualizan los datos primarios, se actualizan todos los datos derivados correctamente.

- *¿Se tiene en cada momento información de los eventos de ratón?*

  Sí se tiene. La clase puntero devuelve correctamente las coordenadas donde se produjo un evento así como el botón con el que se produjo.

- *¿Se mueven correctamente las fichas por la superficie?*

  Sí se mueven. El efecto de arrastrado se creó y optimizó para obtener tiempos de respuesta adecuados.

- *¿Responde correctamente el sistema a todos los eventos que se producen?*

  Sí responde. Se realizaron infinidad de pruebas comprobando que el sistema hace lo que debe hacer dependiendo del lugar en el que se produjera el evento. También se comprobó que no hiciera cosas que no debía.

- *¿Los sonidos se reproducen correctamente?*

  Sí se reproducen. Teniendo además las posibilidad de silenciarlos todos. Los sonidos se reproducen cuando se produce su evento asociado. La banda sonora una vez terminada, vuelve a iniciarse sin saltos, de manera fluida.

- *¿Se muestran los textos correctamente?*

  Sí se muestran. Se muestran las estadísticas de los jugadores, posicionadas en su sitio, así como los números del cronómetro y el mensaje informando del siguiente jugador en el cambio de turno.

- *¿Se puede ordenar el soporte al gusto del jugador?*

  Sí se puede. Se probaron distintas formas de hacerlo y tras conseguirlo se probó que no hubiese problemas de perdidas de fichas ni problemas al insertar ni al extraer.

- *¿Se roba al hacer clic sobre el montón de fichas?*

  Sí. También se roba pulsando la tecla *r*.

- *¿Se comprueba y se pasa turno al hacer clic sobre el botón comprobar y pasar?*

  Sí. También es posible pasar turno pulsando la tecla *s*.

- *¿Se realizan correctamente las comprobaciones y se hace lo que hay que hacer para cada caso?*

  Sí, se comprueba correctamente y se hace lo que hay que hacer. Se ha probado para cada caso con un tablero válido y otro inválido repetidas veces. Los caso son:

  - Tiempo agotado: En el caso válido se roba una ficha, mientras que en el caso inválido se roban tres. Por último se vuelve deja todo como estaba al comienzo del turno.
  - Se pulsó robar: En el caso válido se roba una ficha, mientras que en el inválido se roban tres. Por último se vuelve deja todo como estaba al comienzo del turno.
  - Se pulsó comprobar: En el caso válido se actualizan los valores de las variables, mientras que en el caso inválido se roban tres fichas. Si se pulsó comprobar y no había puesto nada, se roba una ficha.

- *¿Se puede salir de la partida en cualquier momento?*

  Sí. Se puede salir de la partida siempre que tenga el control el jugador pulsando la tecla ESC, en las pantallas de cambio de turno no se puede.

- *¿Se muestra la pantalla de confirmación de salida y funciona correctamente?*

  Sí, se muestra y funciona.

- *¿Se actualizan las puntuaciones de los jugadores una vez finalizada la partida?*

  Sí se actualizan. Dependiendo del tipo de partida que sea lo harán o no: si es concurso se actualizarán, mientras que si es partida simple no.

- *¿Se muestra la pantalla de ganador o perdedor?*

  Sí se muestra.

- *¿Consume muchos recursos de CPU la partida?*

  No. Durante la partida el trabajo de la CPU ronda el 5 %.

- *¿Finaliza correctamente la ejecución de la partida con todas las funcionalidades?*

  Sí. Tras la ejecución de la partida el programa finaliza correctamente.

8.5. Incremento 5: Diseño e implementación de los menús

- *¿Se visualizan correctamente los menús?*

  Sí. Los menús están correctamente posicionados. Además al pasar el ratón por encima de las opciones de los menús estas cambian de imagen y se muestra una pequeña descripción de lo que hace cada una.

- *¿Se escuchan los sonidos?*

  Sí. La banda sonora comienza al inicio de la ejecución del programa, así como los efectos suenan al elegir una opción. Además es posible silenciarlo todo.

- *¿Se captan correctamente los eventos?*

  Sí. Tanto los de movimiento como los de pulsación del ratón, ya que en los menús el teclado no tiene asignado utilidades.

- *¿Conducen las opciones a las secciones correctas?*

  Sí.

- *¿Es posible volver al menú principal desde cualquier submenú?*

  Sí.

8.6. Incremento 6: Diseño e implementación del gestor de partida

- *¿Se ve correctamente la pantalla de selección de jugadores?*

  Sí. Todas las imágenes y textos aparecen correctamente posicionados.

- *¿Se pueden elegir correctamente el número de participantes en una partida?*

  Sí se puede. El sistema no deja que este número sea menor que dos ni mayor que cuatro. Se intentó acceder con números fuera de este rango y el sistema siempre respondió, sin dar lugar a fallos de ejecución ni efectos secundarios.

- *¿Se pueden elegir los jugadores correctamente?*

  Sí se puede. Se probó a insertar jugadores con nombres válidos e inválidos, con distintas longitudes y siempre se devolvió una respuesta correcta.

- *¿Capta correctamente los eventos?*

  Sí los capta. Tanto los de ratón para confirmar o cancelar, como los de teclado para introducir el nombre de los jugadores.

- *¿Se puede salir del gestor de partida en cualquier momento?*

  Sí. Haciendo clic sobre el botón cancelar se vuelve al menú principal.

- *¿Se reproducen los sonidos?*

  Sí. Tanto la banda sonora como los efectos.

- *¿Se crea correctamente una partida simple?*

  Sí se crea. Primero aparece la pantalla de selección de jugadores, se elige el número de participantes, se inserta el nombre del jugador y a continuación el sistema crea tantos robots como participantes haya elegido el jugador menos uno. Se lleva a cabo la partida y una vez finalizada, al tratarse de una partida simple, no se guardan puntuaciones.

- *¿Se crea correctamente un concurso?*

  Sí se crea. Primero aparece la pantalla de selección de jugadores, se elige el número de participantes, se insertan tantos jugadores como se haya elegido en el paso anterior y se juegan tantas partidas como participantes haya. Al final de cada partida se actualizan las puntuaciones de los jugadores.

8.7. Incremento 7: Diseño e implementación del gestor de jugadores

- *¿Se ve correctamente la pantalla de estadísticas de los jugadores?*

  Sí se ve. En la pantalla aparecen de forma ordenada y lógica toda la información referente al jugador seleccionado. En caso de que no haya ningún jugador en el sistema, se muestra un mensaje informando de ello.

- *¿Se puede navegar por la pantalla para elegir distintos jugadores?*

  Sí se puede. Mediante unas flechas de dirección se puede ir moviendo hacía delante y hacía atrás. Las flechas responden correctamente a los eventos de ratón.

- *¿Se puede volver al menú principal desde esta pantalla?*

Sí se puede. Haciendo clic en el botón cancelar.

- *¿Se reproduce el sonido?*

  Sí.

- *¿Se ve correctamente la pantalla para borrar jugadores?*

  Sí se ve. En la pantalla aparece el nombre del jugador seleccionado para borrar y un mensaje que ofrece las instrucciones para borrar al jugador seleccionado.

- *¿Se puede navegar por la pantalla para elegir distintos jugadores?*

  Sí se puede. Mediante unas flechas de dirección se puede ir moviendo hacía delante y hacía atrás. Las flechas responden correctamente a los eventos de ratón.

- *¿Se borran correctamente los jugadores del sistema?*

  Sí se borran. Al hacer clic en el botón confirmar, se elimina del sistema el jugador seleccionado y se vuelve al menú principal.

- *¿Se puede volver al menú principal desde esta pantalla?*

  Sí se puede. Haciendo clic en el botón cancelar.

- *¿Se reproduce el sonido?*

  Sí.

8.8. Incremento 8: Diseño e implementación de los sistemas expertos

- *¿Se colocan correctamente los conjuntos y las escaleras en el tablero?*

  Sí se colocan. Los robots comprueban y extraen correctamente todos los conjuntos o escaleras que poseen y los colocan de forma válida y ordenada en el tablero.

- *¿Se colocan correctamente las fichas sueltas en el tablero?*

  Sí se colocan. Los robots comprueban todos los conjuntos que hay en el tablero comparándolos con las fichas que tienen en sus soportes. si es necesario modifican la posición de los conjuntos en el tablero, para que sigan siendo válidos.

- *¿Se pierden fichas cuando los robots toman el control de la partida?*

  No se pierden. Es uno de los problemas más costosos de resolver, ya que he tenido que realizar numerosas pruebas y trazas a los algoritmos para poder averiguar donde se perdían las fichas, pero finalmente he solucionado el problema en el 100 % de los casos.

- *¿Se ordenan correctamente los soportes con las funciones de ordenar por color y ordenar por número?*

  Sí se ordenan.

- *¿Se puede ordenar por color y por número el soporte de un jugador durante la partida?*

  Sí se puede. Haciendo clic en el botón Ordenar número o bien pulsando la tecla *n*, se ordena el soporte por número. Si se hace clic en el botón Ordenar color o bien se pulsa la tecla *c*, se ordena el soporte por color.

- *¿Siguen los robots las reglas de juego igual que los jugadores?*

  Sí las siguen. A la hora de pasar turno se comprueba el tablero y el soporte al igual que se hace con los jugadores.

- *¿Los sistemas expertos tienen el comportamiento esperado?*

  Sí lo tienen. Además la función que elige aleatoriamente un sistema experto para cada robot, funciona correctamente y hace que no se sepa que inteligencia tiene cada robot.

Capítulo 9

Conclusiones

En este capítulo comentaré las conclusiones e impresiones que he tenido durante la realización de mi proyecto final de carrera.

Comenzaré diciendo que el proyecto me ha ocupado más tiempo del que esperaba en un principio. Me encontré con muchos problemas y muchas dudas las cuales me tuvieron bloqueado en distintas fases del proyecto. Aún así, estoy satisfecho con el trabajo realizado.

Podemos decir que el proyecto goza de una buena calidad, intentando siempre cumplir los objetivos planificados, obteniendo un software sencillo y fácil de utilizar. Personalmente pienso que es algo esencial en un programa de este tipo, el cual la gente usará para divertirse en sus ratos libres. Un punto a favor es el poder manejar todo el juego con el ratón ya que es un sistema más cómodo para navegar que el teclado.

En cuanto a las cosas que he aprendido, evidentemente, he aprendido a hacer un proyecto más o menos complejo: crear calendarios, estudiar posibles ramas de desarrollo, buscar las herramientas adecuadas, realizar trabajos de ingeniería del software, documentar código, organizarlo, etc. Durante los años de la carrera he realizado diversas prácticas y he realizado trabajos más o menos complejos, pero nada similar a lo que significa realizar el Proyecto Final de Carrera. Me hubiese gustado tener mayor conocimiento en el área de ingeniería de software, y así hubiese ahorrado bastante tiempo solucionando problemas de planificación y diseño. A pesar de todo, pienso que estoy en condiciones de realizar un proyecto software con bastantes posibilidades de éxito.

Gracias a este proyecto he aprendido a utilizar herramientas con LaTeXy en particular la plantilla realizada por un compañero de la Universidad de Cádiz [9], la cual me ha ahorrado muchísimo tiempo y esfuerzo, además de la gran calidad de presentación que proporciona.

He profundizado en el manejo y configuración de *Doxygen*, pienso que otra herramienta imprescindible a la hora de documentar el código de una aplicación ya que prácticamente sin ningún esfuerzo se obtiene una documentación de gran calidad.

He aprendido a utilizar en profundidad la librería *libSDL*, gracias al manual del compañero *Antonio García Alba* [2],así como a manejar el editor de textos *emacs*, el cual posee infinidad de opciones.

A nivel personal, he aprendido a imponerme un ritmo de trabajo, seguir un horario, cumplir unas fechas de entrega, sin que nadie este supervisando mi trabajo o si estoy cumpliendo con lo dicho anteriormente.

9.1. Posibles ampliaciones

En este apartado comentaré posibles mejoras que puedan realizarse al proyecto.

1. **Juego en red**: Debido a la naturaleza del juego, pienso que sería una buena opción incluir esta característica en versiones futuras del mismo, ya que se podría llegar a conseguir jugar a *FreeRummi* con personas de cualquier parte del mundo.

2. Crear un **sitio web** en el que añadir records, puntuaciones y experiencias del juego.

3. Incluir el juego en **tecnologías móviles**: debido al gran impacto que están teniendo las aplicaciones en teléfonos móviles, sería una buena opción que el juego estuviese operativo para estas plataformas, incluyendo también en esta versión el juego en red.

4. **Mejorar los elementos gráficos**, ya que no soy un experto en diseño gráfico, la interfaz gráfica del juego aunque es admisible, no deja de ser mejorable.

Apéndice A

Manual de instalación

A.1. Obtener FreeRummi: juego de fichas numeradas

Lo primero que debemos hacer es obtener una versión de *FreeRummi*.

Lo podemos encontrar en la página del proyecto en la forja de rediris [14] o usando la herramienta *Subversion*, escribiendo en la consola el siguiente comando:

```
svn checkout https://forja.rediris.es/svn/freerummi
```

A.2. Instalación

Para la instalación necesitaremos tener instaladas unas librerías adicionales.

Si no tenemos instaladas las librerías *SDL*, tendremos que instalarlas. En este caso basta con instalar la librería y las librerías adicionales *SDL_mixer*, *SDL_image* y *SDL_ttf* mediante el gestor de paquetes o la línea de comandos. En un entorno *debian* los comandos a ejecutar serían:

```
sudo atp-get install libsdl1.2debian
sudo atp-get install libsdl-ttf2.0-0-dev
sudo atp-get install libsdl-image1.2-dev
sudo atp-get install libsdl-mixer1.2-dev
```

Una vez instaladas nos dirigimos al directorio *freerummi*:

```
cd /freerummi
```

Y a continuación compilamos con “make”:

```
make
```

Si la compilación ha tenido éxito nos generará un fichero ejecutable llamado *freerummi*, por tanto sólo tendremos que ejecutarlo.

```
./freerummi
```

O bien haciendo doble clic en el entorno gráfico.

Apéndice B

Manual de usuario

Bienvenido al manual de usuario de *FreeRummi*, en las siguientes páginas aprenderá las reglas de este juego así como a moverse por los menús del mismo.

B.1. El juego

Las reglas de *FreeRummi* están extraídas del manual oficial de *Rummikub*® [10].

B.1.1. Objetivo del juego

Ser el primero jugador en colocar todas las fichas de soporte y obtener la mayor puntuación al sumar las fichas que queden en los soportes de los otros jugadores.

B.1.2. Las reglas

1. Para empezar a jugar, cada jugador debe tener combinaciones que sumen **como mínimo** 30 puntos (determinado por la suma del valor de cada ficha). Si un jugador no puede empezar a jugar porque no tiene los 30 puntos, o porque decide que no quiere hacerlo, deberá coger una ficha del montón. Con esto termina su turno. El juego continua hacía el jugador de la derecha.

2. Los jugadores deben seguir cogiendo fichas del montón, en cada uno de sus turnos, hasta que obtengan la combinación o combinaciones que sumen 30 puntos (mínimo) y que les permita empezar a jugar. Cuando lo consigan, deben colocar sus combinaciones sobre la mesa. Una vez colocadas en la mesa, las combinaciones de números pertenecen a todos los jugadores y cada uno es libre de modificarlas o formar otras nuevas con fichas de sus soportes, según les convenga.

3. Los jugadores tiene un tiempo límite de 1 minuto por turno. Si pasado el minuto, no han conseguido arreglar con éxito las fichas de la mesa, deben devolverlas a su posición original sobre el tablero y coger tres fichas más del montón como penalización.

4. El juego continua hasta que uno de los jugadores vacíe por completo su soporte. Los demás jugadores suman los puntos de sus soportes.

B.1.3. Manipulación

1. La manipulación es la parte más interesante de *FreeRummi*. Consiste en añadir o alterar las combinaciones previamente jugadas, de forma que le permita al jugador colocar sobre la mesa el máximo de fichas de su soporte.

2. Las fichas pueden ser manipuladas en cualquiera de las siguientes maneras, siempre que, al final de cada turno, solo haya combinaciones válidas sobre la mesa y no queden fichas sobrantes.

1. **Añadir una o más fichas del soporte para formar una escalera:**

   Dados el siguiente soporte B.1, y el siguiente tablero B.2:

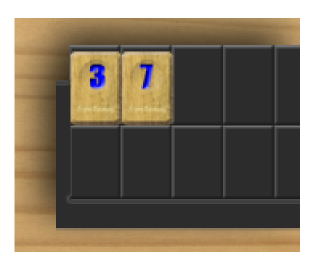


Figura B.1: Caso 1: Soporte antes de la manipulación

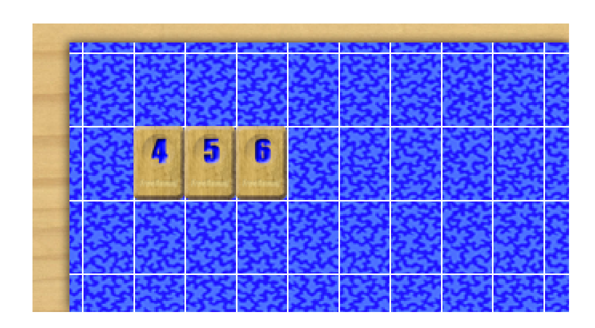


Figura B.2: Caso 1: Tablero antes de la manipulación

   El 4, 5 y 6 azules están sobre el tablero; el jugador añade 3 y un 7 azules de su soporte, teniendo entonces B.3:

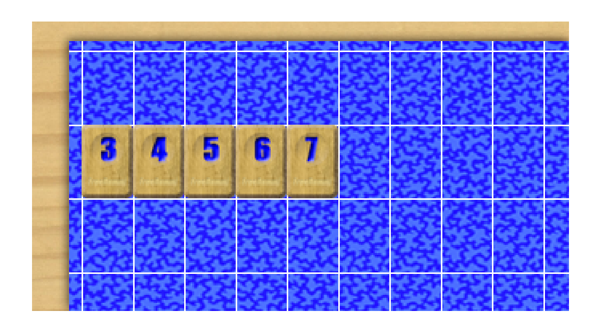


Figura B.3: Caso 1: Tablero después de la manipulación

2. **Quitar una cuarta ficha de una serie y utilizarla para crear una nueva combinación:**

   Dados el siguiente soporte B.4, y el siguiente tablero B.5:

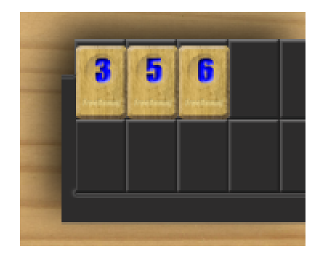


Figura B.4: Caso 2: Soporte antes de la manipulación

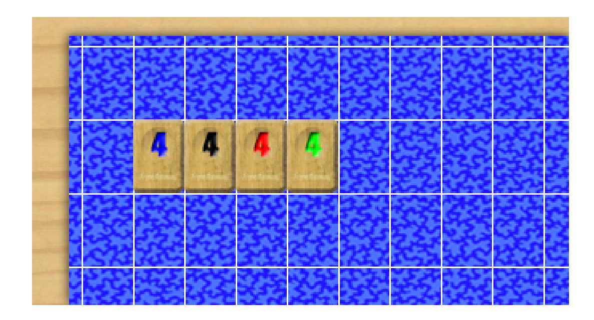


Figura B.5: Caso 2: Tablero antes de la manipulación

Falta una ficha en el soporte para crear una escalera azul. El jugador coge un 4 azul del conjunto de cuatro que hay sobre el tablero y coloca la escalera: 3, 4, 5 y 6 azules, teniendo entonces B.6:

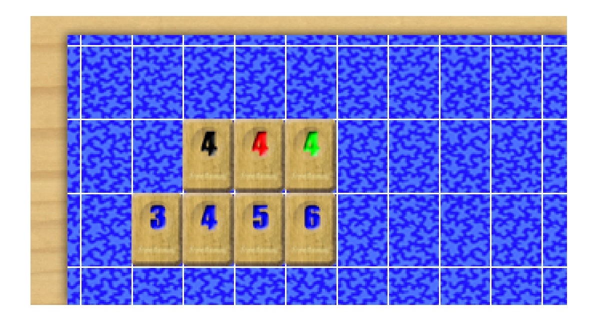


Figura B.6: Caso 2: Tablero después de la manipulación

3. **Añadir una cuarta ficha a una combinación y quitar otra para crear una combinación diferente:**

   Dados el siguiente soporte B.7, y el siguiente tablero B.8:

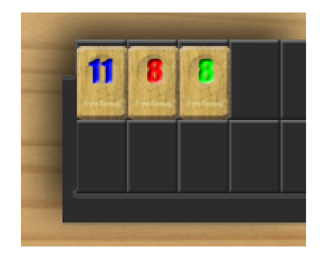


Figura B.7: Caso 3: Soporte antes de la manipulación

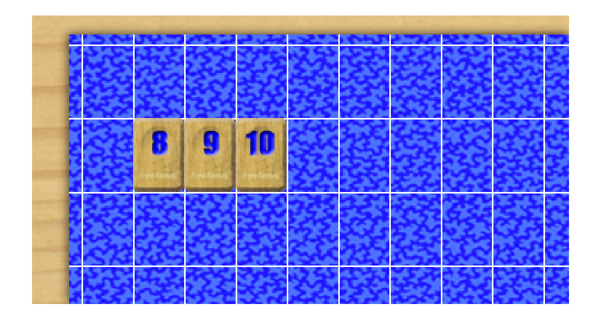


Figura B.8: Caso 3: Tablero antes de la manipulación

El jugador añade el 11 azul a la escalera y utilizar el 8 azul para formar un nuevo conjunto, teniendo entonces B.9:

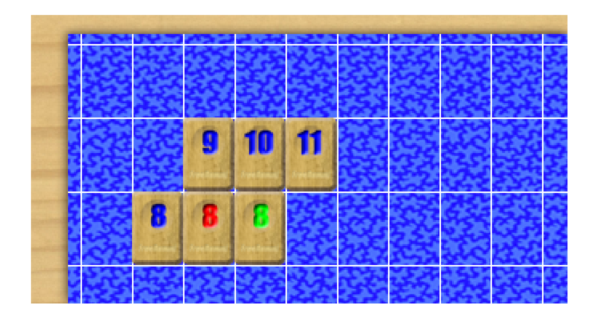


Figura B.9: Caso 3: Tablero después de la manipulación

4. **Dividir una escalera:**

   Dados el siguiente soporte B.10, y el siguiente tablero B.11:

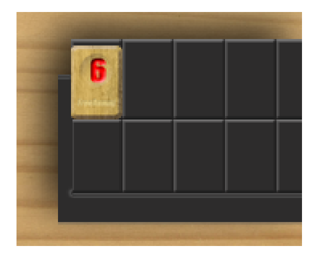


Figura B.10: Caso 4: Soporte antes de la manipulación

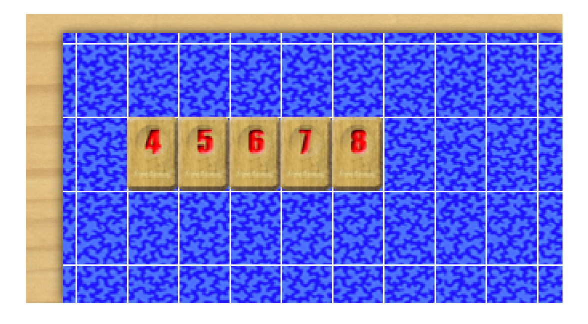


Figura B.11: Caso 4: Tablero antes de la manipulación

El jugador divide la escalera y utiliza el 6 rojo para formar dos escaleras nuevas, teniendo entonces B.12:

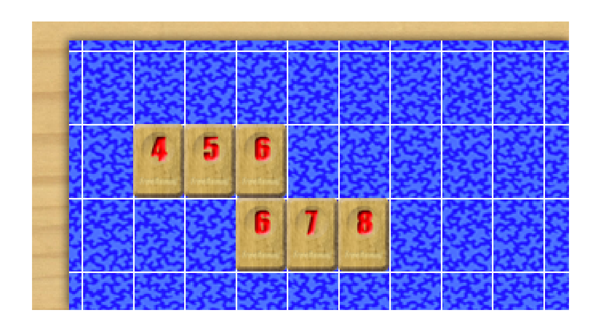


Figura B.12: Caso 4: Tablero después de la manipulación

5. **División combinada:**

Dados el siguiente soporte B.13, y el siguiente tablero B.14:

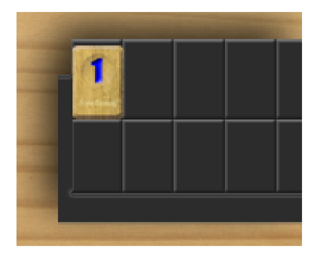


Figura B.13: Caso 5: Soporte antes de la manipulación

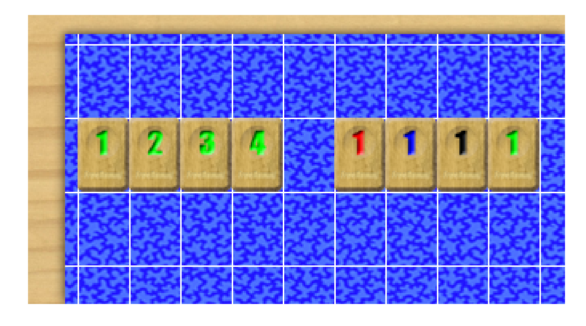


Figura B.14: Caso 5: Tablero antes de la manipulación

El jugador coloca el 1 azul de su soporte con el 1 verde de la escalera y con el 1 rojo del conjunto para crear un nuevo conjunto, teniendo entonces B.15:

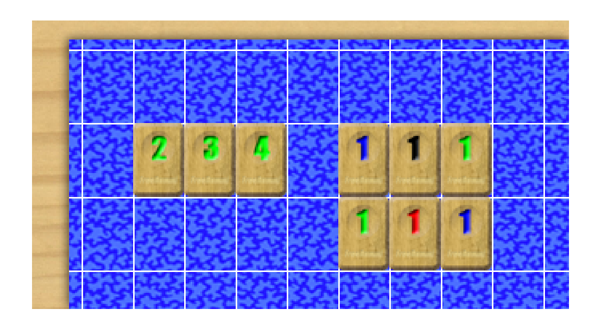


Figura B.15: Caso 5: Tablero después de la manipulación

6. **División múltiple:**

   Dados el siguiente soporte B.16, y el siguiente tablero B.17:

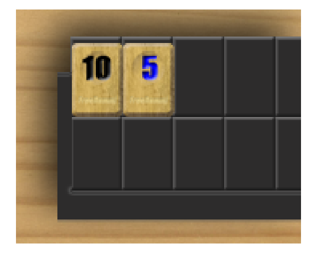


Figura B.16: Caso 6: Soporte antes de la manipulación

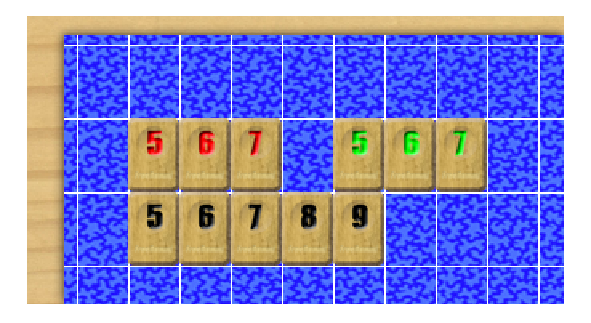


Figura B.17: Caso 6: Tablero antes de la manipulación

El jugador manipula las tres combinaciones existentes sobre la mesa y utiliza el 10 negro y el 5 azul de su soporte para crear tres conjuntos y una escalera nuevas B.18:

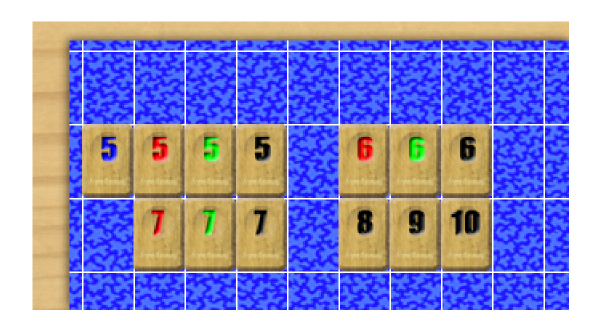


Figura B.18: Caso 6: Tablero después de la manipulación

B.1.4. El comodín

1. Hay dos comodines en el juego que se pueden utilizar como cualquier ficha, y en cualquier combinación, independientemente de los números y el color. En una combinación de números, el valor del comodín es el de la cifra a la cual representa.

2. Cualquier jugador puede retirar el comodín de una combinación que haya en el tablero y sustituirlo en su turno con una ficha del mismo valor numérico y color al que representa.

3. La ficha utilizada para sustituir al comodín puede proceder tanto del soporte de un jugador como del tablero. En el caso de un conjunto de 3, el comodín puede ser sustituido por una ficha de cualquiera de los colores que falten.

4. Al sustituir el comodín por la ficha correspondiente, el jugadores deberá utilizarlo en ese mismo turno, y no podrá colocarlo en su soporte para futuras jugadas.

5. Cualquier combinación que tenga un comodín puede ser dividida, además de poder añadir o quitar fichas de la misma.

6. Al final de la partida, si queda algún comodín sobre el soporte de cualquier jugador, su valor es de 30 puntos.

B.1.5. Puntuación

1. Después de que un jugador haya vaciado su soporte, los demás jugadores suman los puntos de sus soportes y se los anotan en negativo. La puntuación del ganador será la suma de los puntos de las fichas que tengan los otros jugadores en sus soportes.

2. A la hora de comprobar las cifras, puede servir de ayuda saber que los puntos positivos deberían ser iguales al total de los puntos negativos de cada ronda.

3. En el raro supuesto de que se utilicen todas las fichas del montón antes de que algún jugador vacíe su soporte, el jugador que tenga meno puntuación al sumar las fichas de su soporte será el ganador. El resto de los jugadores suman el valor de las fichas de sus soportes y lo resta al total del ganador. Este resultado es el que deberán anotar en negativo. La puntuación del ganador será la suma de los puntos negativos de los demás jugadores.

	Jugador A	Jugador B	Jugador C	Jugador D
Partida 1	+24	-5	-16	-3
Partida 2	-6	-11	+22	-5
Total	+18	-16	+6	-8

Tabla B.1: Ejemplo de puntuación

B.1.6. Estrategia

El principio de una ronda de *FreeRummi* puede parecer un poco lenta pero, a medida que se añaden combinaciones sobre la mesa, las manipulaciones que se pueden hacer son cada vez mayores. Al principio de una partida, a veces es conveniente no colocar todas las fichas posibles sobre la mesa y dejar a los otros jugadores que abran el juego, lo que permite tener más oportunidades de modificar las combinaciones existentes.

En ocasiones, es útil guardar la cuarta ficha de un conjunto o escalera y colocar únicamente tres, de forma que en el siguiente turno pueda colocarse una ficha en lugar de cogerla del montón.

También a veces es interesante no jugar el comodín y tenerlo en reserva, pero atención, recuerda que vale 30 puntos si se queda en tu soporte al final de la partida.

B.2. El menú principal

La imagen que puede observarse en la figura B.19 muestra el **menú principal** de *FreeRummi*.



Figura B.19: Menu principal de FreeRummi

El **menú principal** contiene las opciones para acceder a las distintas funcionalidades del juego:

- **Jugar:** Podrás jugar una partida simple contra la máquina o participar en un concurso contra otros amigos.

- **Jugadores:** Podrás ver las estadísticas de los jugadores (puntuaciones, porcentajes, rangos), así como eliminarlos.

B.3. El menú Jugar

En la figura B.20 pueden observarse las distintas opciones que presenta el **menú Jugar**.



Figura B.20: Menu Jugar de FreeRummi

- **Simple:** Juega una partida a *FreeRummi* contra el ordenador. En este tipo de partida no se tiene en cuenta la puntuación final que se obtenga. Estas partidas pueden servir de entrenamiento para adquirir experiencia.

- **Concurso:** Participa en un tornero de *FreeRummi* contra tus amigos. En este tipo de partida estará compuesto por tantas rondas como jugadores seáis y se tendrá en cuenta la puntuación.

B.3.1. Pantalla de selección de jugadores

La pantalla de **selección de jugadores** está dividida en dos partes como puede observarse en la figura B.21:



Figura B.21: Pantalla de selección de jugadores

1. **Selección del número de jugadores:** Aquí se escogerá el número de participantes que jugarán en la partida. Este número estará comprendido entre dos y cuatro.

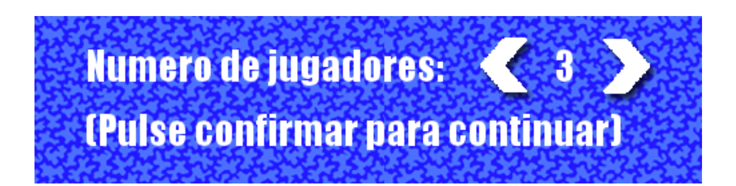


Figura B.22: Selección del número de jugadores

2. **Selección de jugadores:** Aquí se irán introduciendo los nombres de los jugadores que participarán. Recuerda que el nombre solo puede tener ocho caracteres.

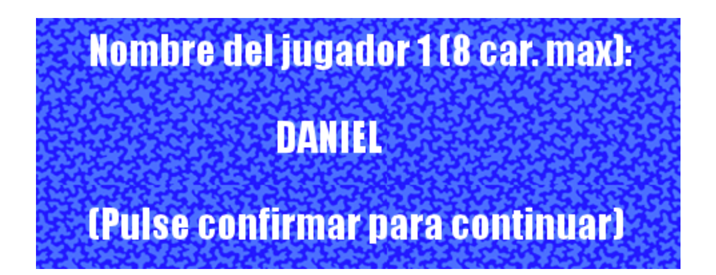


Figura B.23: Selección de los jugadores

B.3.2. Pantalla de partida

En la figura B.24 se pueden observar los distintos elementos que componen la partida de *FreeRummi*:

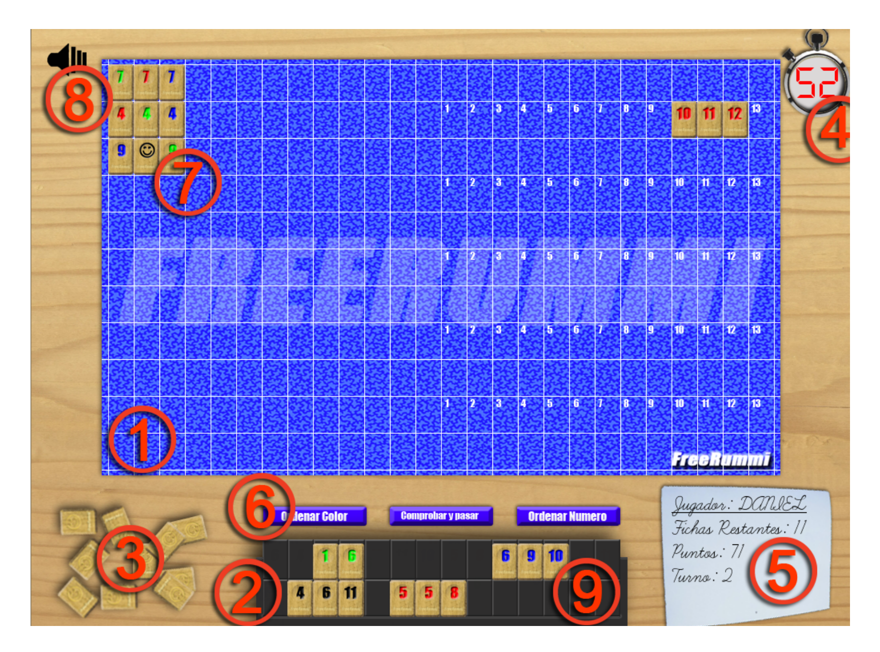


Figura B.24: Pantalla de una partida

1. **Tablero:** Es el lugar donde se colocarán todas aquellas combinaciones de fichas que los jugadores puedan formar. Las fichas que están en el tablero pueden ser manipuladas por todos los jugadores.
2. **Soporte:** Es el lugar donde se guardan las fichas de cada jugador. Estas fichas no deben ser vistas por nadie más que el propietario de las mismas. El jugador es libre de moverlas dentro de su soporte para tenerlas ordenadas a su gusto.
3. **Montón:** En él están las fichas restantes que no están ni en el tablero ni en los soportes de los jugadores. Cuando un jugador no tiene combinaciones para poner o, simplemente decide no ponerlas, tiene que sacar una ficha del montón la cual se insertará en su soporte.
4. **Cronometro:** Indica el tiempo restante en el turno de un jugador. La duración de un turno son 60 segundos.
5. **Información del jugador:** Indica de qué jugador es el turno, así como las fichas que tiene en su soporte, su puntación y el número de turno que va jugando.
6. **Botones de acciones:** Con estos botones el jugador podrá ordenar por colores su soporte, comprobar el tablero y pasar el turno al siguiente jugador y ordenar por números su soporte, respectivamente.
7. **Grupos de fichas:** Son fichas colocadas en el tablero formando una escalera o un conjunto. Estos grupos pueden ser manipulados por cualquier jugador.
8. **Sonido ON/OFF:** Con este botón se puede silenciar el juego o hacerlo volver a sonar.
9. **Fichas restantes:** Son las fichas que quedan en el soporte de un jugador, de las cuales deberá deshacerse antes que cualquier otro jugador si quiere ganar.

B.4. El menú Jugadores

Las opciones disponibles en este menú pueden verse en la figura B.25:



Figura B.25: Menú Jugadores de FreeRummi

- **Ver:** Observa tus estadísticas y las de los demás jugadores de *FreeRummi*.
- **Borrar:** Elimina un jugador que ya no interese del sistema.

B.4.1. Pantalla Ver

La pantalla **Ver** presenta el aspecto de la figura B.26:



Figura B.26: Pantalla Ver de FreeRummi

Esta pantalla presenta las estadísticas de los jugadores, las cuales son:

1. Puntuación general
2. Partidas ganadas
3. Partidas perdidas
4. Partidas abandonadas
5. Porcentaje de partidas ganadas
6. Porcentaje de partidas perdidas
7. Porcentaje de partidas abandonadas
8. Rango

B.4.2. Pantalla Borrar

Esta pantalla presenta el siguiente aspecto B.27:

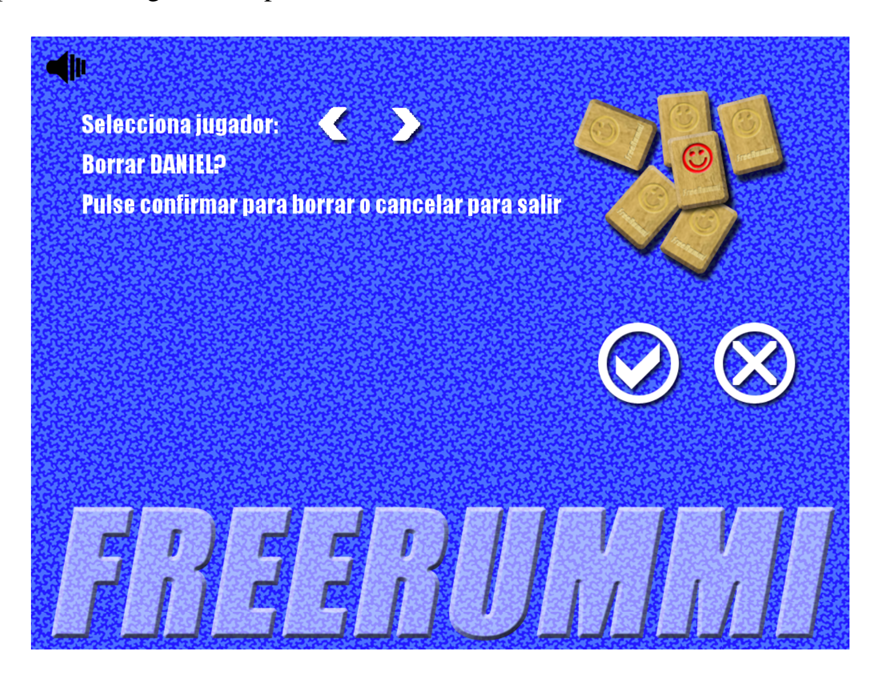


Figura B.27: Pantalla Borrar de FreeRummi

B.5. Controles y atajos de teclado

Para acabar el manual de usuario, voy a comentar cómo se puede interactuar con el sistema. Esta aplicación está diseñada para ser totalmente controlada con el ratón, aunque en el modo partida también existen atajos de teclado que harán más cómodo el juego. A continuación se explican más detalladamente los controles de *FreeRummi*

B.5.1. Navegar por los menús

Para navegar por los menús de *FreeRummi*, tan solo tiene que situarse encima de la opción que desee y hacer clic con el botón izquierdo del ratón.

Si está dentro de un menú y desea volver atrás, solo tiene que hacer clic izquierdo en la opción **Volver**. En las pantallas **Selección de jugadores**, **Ver** y **Borrar** haga clic izquierdo en el botón *confirmar* B.28 para llevar a cabo la acción deseada o bien haga clic izquierdo en el botón *cancelar* B.29 para volver al menú principal.



Figura B.28: Botón confirmar



Figura B.29: Botón cancelar

Para activar o desactivar el sonido, hacer clic con el botón izquierdo en el altavoz situado en la parte superior izquierda de la pantalla.

B.5.2. Controles en la partida

A continuación se enumeran los controles y atajos de teclado que posee la partida de *FreeRummi*:

- **Robar Ficha:** Hacer clic izquierdo en el área del montón de fichas o pulsar la tecla *R*
- **Pausar el juego:** Hacer clic izquierdo en el área del cronometro o pulsar la tecla *P*
- **Abandonar la partida:** Pulsar la tecla *ESC*
- **Ordenar el soporte por color:** Hacer clic izquierdo en el botón *Ordenar Color* o pulsar la tecla *C*
- **Ordenar el soporte por número:** Hacer clic izquierdo en el botón *Ordenar Número* o pulsar la tecla *N*
- **Comprobar el tablero y pasar turno:** Hacer clic izquierdo en el botón *Comprobar y pasar* o pulsar la tecla *S*
- **Activar/Desactivar el sonido:** Hacer clic izquierdo en el altavoz o pulsar la tecla *M*.

Bibliografía

[1] Simple directmedia layer. http://www.libsdl.org/, Consultado Abril 2011.

[2] Antonio García Alba. *Tutorial de libSDL para la programación de videojuegos*. http://softwarelibre.uca.es/node/890, 2008.

[3] cplusplus.com. C++ reference. http://www.cplusplus.com/reference/.

[4] Ministerio de educación. Banco de imagenes y sonidos del Instituto de tecnologías educativas. http://recursostic.educacion.es/bancoimagenes/web/.

[5] Gerardo Aburruzaga García Francisco Palomo Lozano, Inmaculada Medina Bulo. *Fundamentos de C++, ISBN 8498280079*. Servicios de Publicaciones de la Universidad de Cádiz, 2º edition.

[6] Joseph Giarratano Gary Riley. *Sistemas expertos: principios y programación, ISBN 9706860592*. Internacional Tompson Editores S.A., 3º edition, 2001.

[7] Jamendo. Musica libre. http://www.jamendo.com/es/.

[8] Glenford J Myers. *The art of software testing, ISBN 0471469122*. Hoboken: John Wiley & Sons, 2º edition, 2004.

[9] Pablo Recio Quijano. Plantilla en Latex para PFC. http://softwarelibre.uca.es:9001/redmine/wiki/pfc/Plantillas_LaTeX_para_PFC, Consultado Abril 2011.

[10] Rummikub®. *Instrucciones de Juego*. Parker®.

[11] Rummikub®. The original Rummikub®. http://www.rummikub.com, Consultado Abril 2011.

[12] Bernardo Cascales Salinas. *LaTeX: una imprenta en sus manos, ISBN 84923895*. Pearson Educación, 2003.

[13] Antonio García Serrano. *Programación de videojuegos con SDL para Windows y Linux, ISBN 8495836084*.

[14] Daniel Chaves Vázquez. Forja de FreeRummi: Juego de fichas numeradas. https://forja.rediris.es/news/?group_id=783.

[15] Wikipedia. Página wiki de Rummikub®. http://es.wikipedia.org/wiki/Rummikub, Consultado Abril 2011.

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.
The “**publisher**” means any person or entity that distributes copies of the Document to the public.
A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.
“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.